BLAISE.Guide d'utilisation

François Delebecque, C. Klimann, Serge Steer

INRIA

CENTRE DE ROCQUENCOURT

Rapports Techniques

N° 56

BLAISE
GUIDE D'UTILISATION

François DELEBECQUE
Carlos KLIMANN
Serge STEER

Juillet 1985

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(3) 954 90 20

B L A I S E

Résumé *: Ce rapport décrit l'usage du système BLAISE, logiciel interactif conçu pour l'Automatique.*

Summary *: This report describes the conversational system BLAISE, a software-package designed for Automatic control.*

BLAISE

-Guide d'Utilisation-

François Delebecque
Carlos Klimann
Serge Steer

Version du 8-8-85

TABLE DE MATIERES

1/_INTRODUCTION

1.1/_GENERALITES.

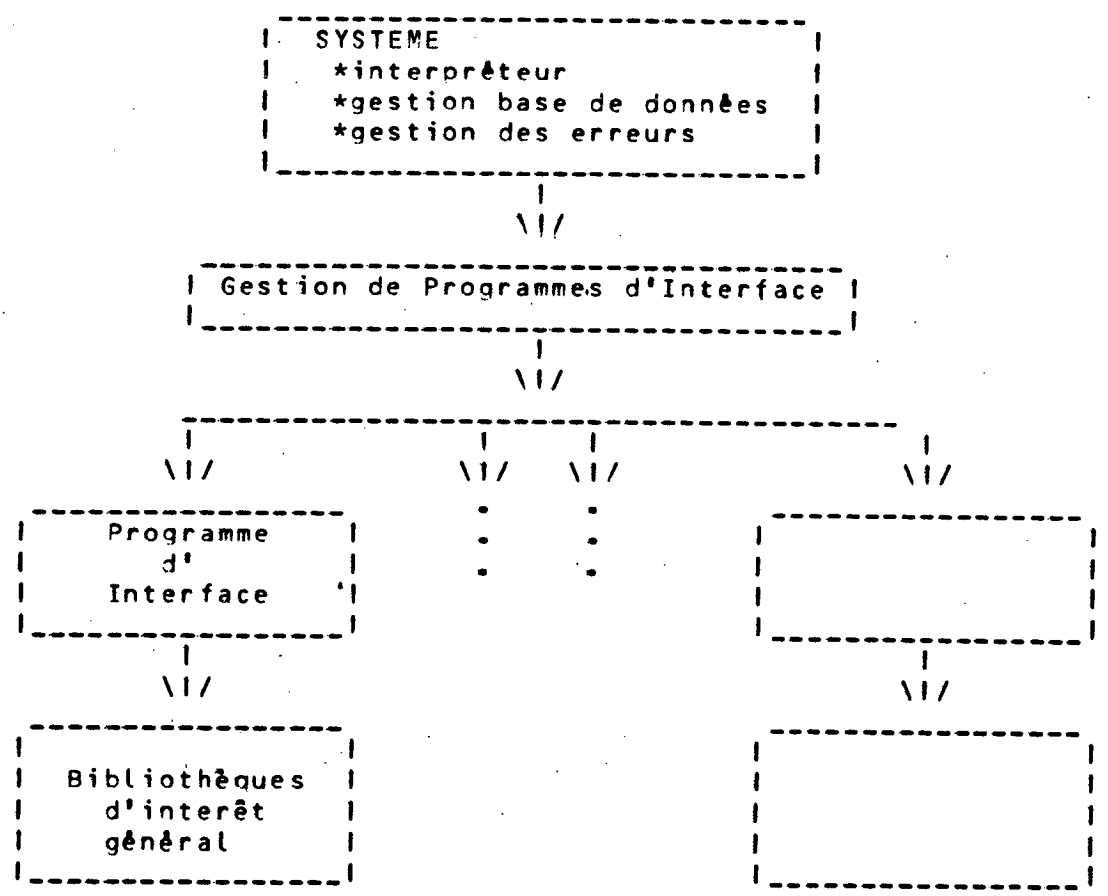
BLAISE est un système interactif conçu pour le développement et la résolution des problèmes relevant de l'Automatique classique. Ce système a été realisé avec l'objectif de fournir aux experts en Automatique un outil puissant de recherche et de développement. Jusqu'à present le système a été porté avec succès sur Honeywell DPS8/MULTICS, VAX/VMS, HP9000 et Gould-X32.

Le système BLAISE est totalement interactif, il reçoit une ligne de commande, la traite immédiatement, le cas échéant imprime le résultat et attend la nouvelle commande. D'autre part on a cherché à rendre le système le plus "ouvert" possible; l'utilisateur peut, soit introduire lui-même une nouvelle commande associée à un ou plusieurs programmes FORTRAN, soit créer des procédures, écrites en langage BLAISE, et directament utilisables de façon interactive. Les possibilités d'extension des fonctionnalités de BLAISE sont donc pratiquement illimitées. Il est clair, cependant que le caractère interactif du système est limité par le temps de calcul nécessaire à l'execution de la commande donnée à BLAISE. D'autre part, BLAISE a des facilités graphiques qui peuvent être de grande utilité dans la visualisation des solutions obtenues.

Un effort particulier a été déployé pour donner au langage BLAISE la plus grande similitude possible avec l'écriture mathématique courante. Cependant la nature des objets de l'Automatique classique (matrices réelles ou complexes, polynômes, matrices polynomiales, etc.) et l'obligation de "linéarité" de l'écriture dans un ordinateur ont obligé parfois à s'éloigner du modèle choisi.

Des efforts particuliers quant à la portabilité ont été faits en écrivant l'ensemble du système en FORTRAN 77 standard. BLAISE se compose d'une partie "système" (environ 15000 lignes de FORTRAN) qui interprète la ligne de commande et gère l'interface avec une bibliothèque de programmes (50000 lignes). La structure modulaire de l'ensemble permet de modifier facilement cette configuration.

Le système BLAISE a été développé à partir du logiciel MATLAB [9a] dont on a conservé en grand partie l'interpréteur. La base de données a, en revenche, été complètement modifiée pour traiter les objets spécifiques de l'Automatique (fonctions non linéaires ou matrices polynomiales, par exemple).



1.2/ MODE DE FONCTIONNEMENT

Après installation dans un site particulier l'usage du système BLAISE se fait généralement en tapant "blaise" à la console. L'utilisateur reçoit alors le message de bienvenue du système BLAISE, puis le "a-vous" (ou "prompt") <>.

BLAISE attend alors la première commande, la traite, renvoie le résultat de la commande et produit de nouveau le "a-vous" <>. Le système est donc totalement interactif. Il conserve en mémoire l'ensemble des variables définies par l'utilisateur lors d'une même session. Ces variables peuvent être des variables numériques ou des macro-instructions (sous-programmes écrits en langage BLAISE). Des commandes spécialisées (voir le paragraphe de la page 17) permettent de sauver l'information d'une session à l'autre.

Typiquement un utilisateur de BLAISE définira ses propres programmes en langage BLAISE (macro-instructions). Les programmes faisant appel aux puissantes capacités numériques de BLAISE, l'utilisateur se trouve affranchi d'un pénible travail de programmation, tout en pouvant traiter des problèmes très particuliers.

1.3/ JEU DE CARACTERES, LIGNES DE COMMANDE, SYNTAXE.

1.3.1/ Caractères:

Le jeu de caractères de BLAISE consiste en 10 chiffres, deux fois 26 lettres et 21 caractères spéciaux. Il faut noter que dans la version actuelle certains caractères sont équivalents et ont la même représentation interne.

CARACTERES	CODES BLAISE
0-9	0-9
a-z ou A-Z	10-35
blanc	36
(	37
)	38
;	39
: ou !	40
+	41
-	42
*	43
/	44
\ ou $	45
=	46
.	47
,	48
' ou "	49
< ou [	50
> ou]	51
%	52

1.3.2/ Lignes de commandes:

Une ligne de commande est une séquence de au plus 80 caractères. Les lignes de continuation sont indiquées par au moins deux points consécutifs en fin de la ligne précédente. Le nombre maximum de caractères d'une commande ne peut pas exceder 1024 caractères. Le caractère blanc est, en général, significatif; par contre, un ou plusieurs blancs consécutifs ont le même sens.

Tout ce qui suit un double "slash" (//) dans une ligne de commande n'est pas pris en compte, et peut servir de commentaire.

1.3.3/ Diagrammes de la syntaxe:

Nous donnons ici une description formelle du langage BLAISE sous la forme de graphes de syntaxe de Wirth [16]. Ce paragraphe, inséré ici à titre de référence, peut être sauté.

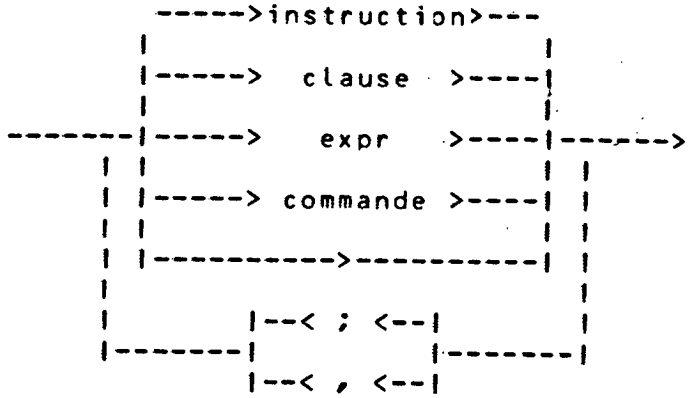
Il y a onze symboles intermédiares:

Ligne, instruction, clause, expression (expr), terme, facteur, nombre, entier (ent), nom, commande et texte.

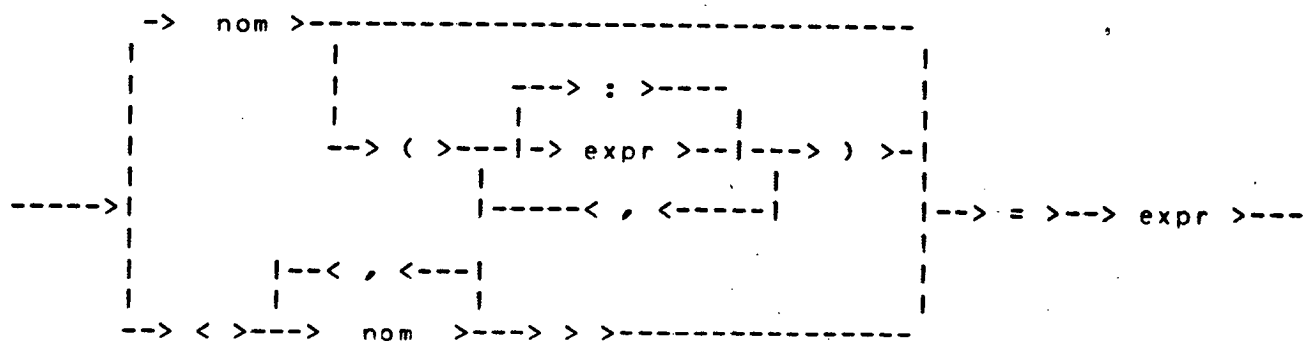
que l'on peut définir à partir du jeu de caractères suivant:

lettres: a-z, A-Z chiffres: 0-9 car: blanc ( ) + : | + - * / \ $ = . , < [>] % quote: ' "

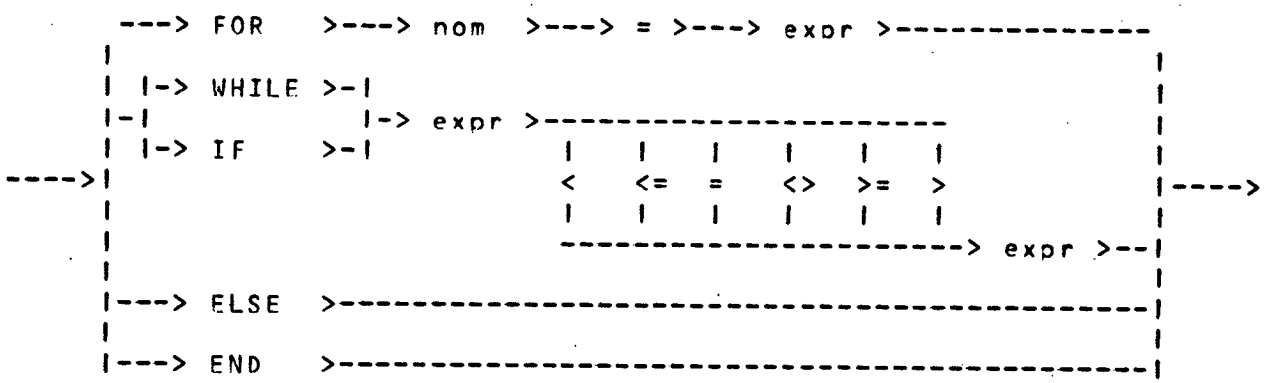
ligne



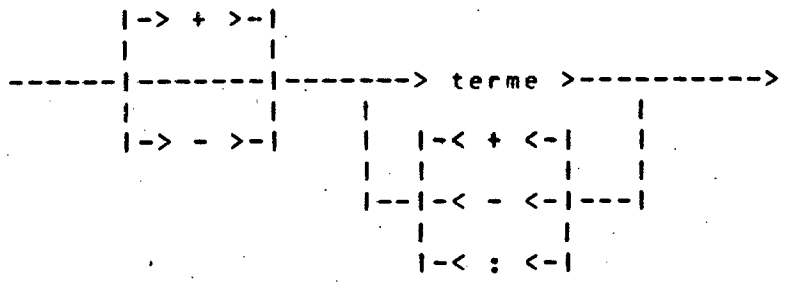
instruction



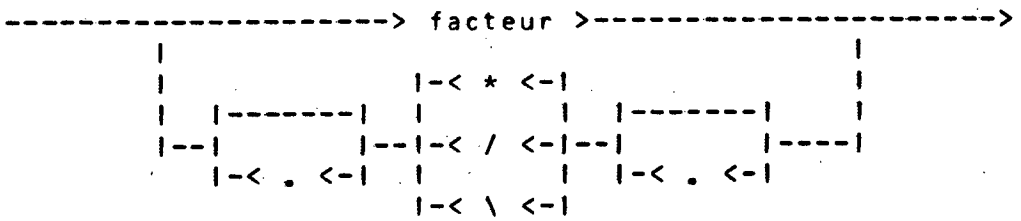
clause



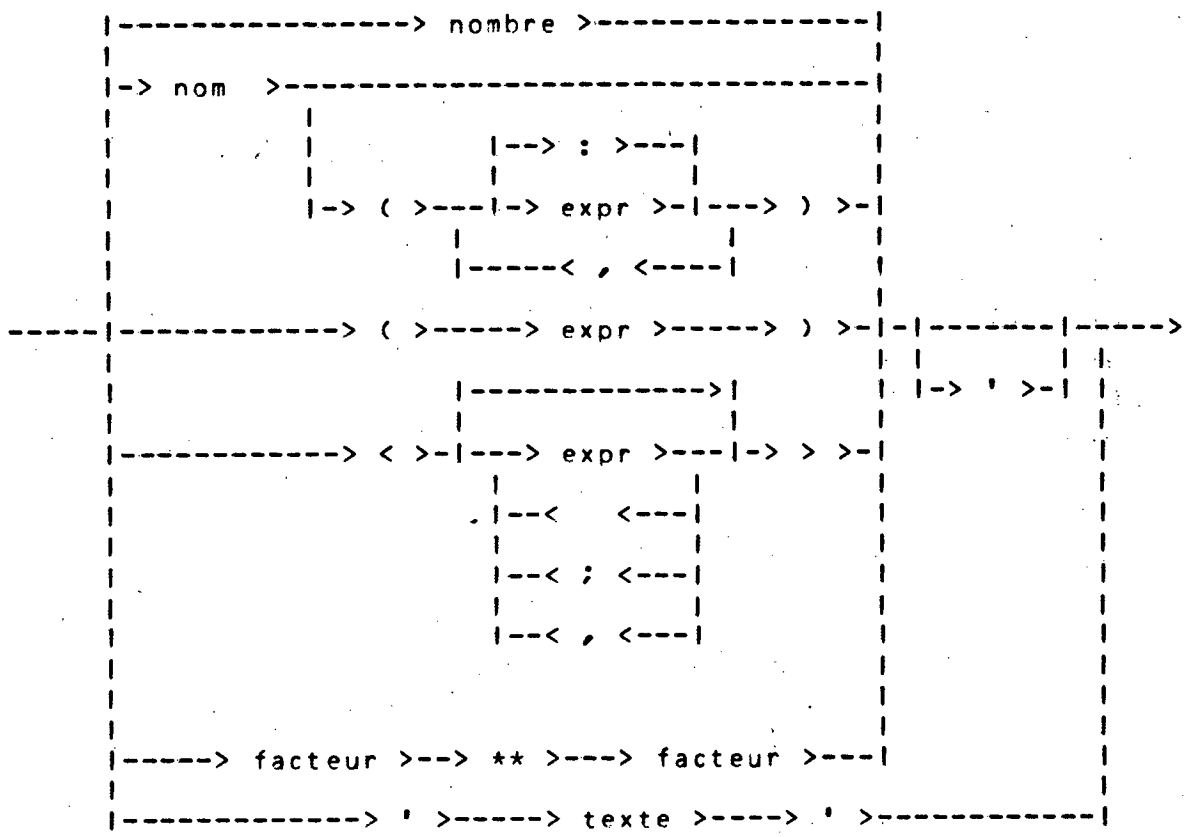
expr



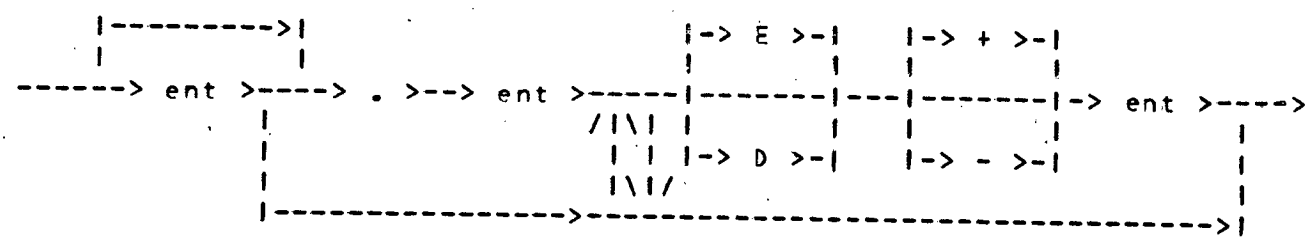
terme



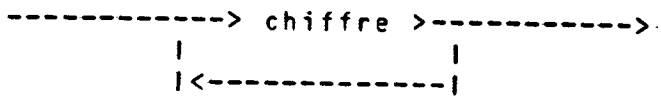
facteur



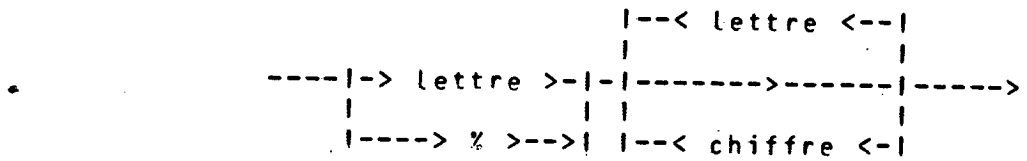
nombre



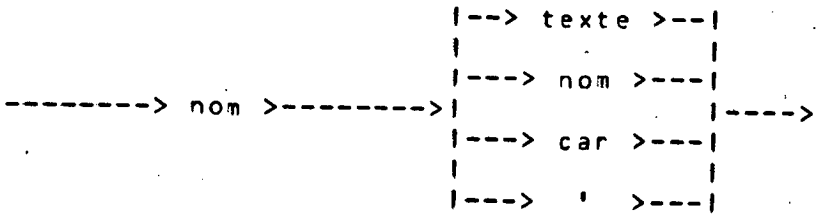
ent



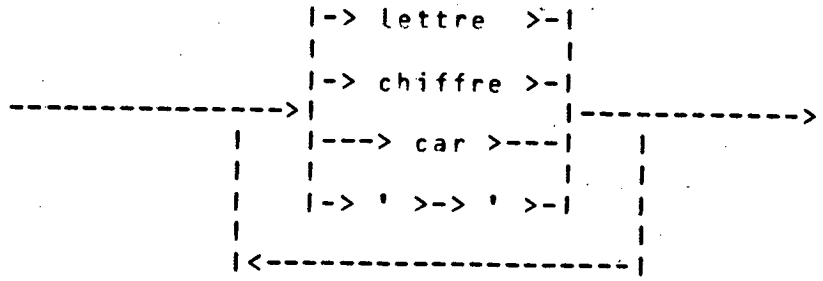
nom



commande



texte



2/ TYPES ET OPERATIONS ELEMENTAIRES.

2.1/ CONSTANTES

2.1.1/ Constantes Scalaires Predéfinies:

Les valeurs de certaines constantes d'emploi fréquent et dont les valeurs sont difficiles a écrire se représentent avec un nom symbolique, ainsi:

%i	racine carrée de -1
%pi	le nombre PI
%e	le nombre e, base des logarithmes naturels
eye	matrice identité
ones	matrice de uns.

eye(n) est la matrice identité d'ordre n. eye(m,n) est une matrice m par n avec des 1 sur la diagonale et 0 ailleurs. eye(a) a la même taille que a. eye sans arguments est la matrice identité dont l'ordre est defini par le contexte. Par exemple, (3 * eye + a) ajoute 3 aux éléments diagonaux de a.

ones est la matrice dont tous les éléments valent 1. ones(n), ones(m,n), ou ones(a) sont valides pour, respectivement, une matrice n par n, n par m ou de même taille que la matrice a.

Pour définir une matrice dont tous le éléments sont nuls on utilisera, par exemple, l'écriture:

```
0 * ones(m, n).
```

2.1.2/ Constantes Scalaires:

Les constantes scalaires se représentent par leur valeur:

```
0    1    1348.45    0.37856912    1+%i    3*%pi    -8.5

1.348E+7 37.85D-2 ...
```

Les constantes que nous venons de décrire ne sont qu'un cas particulier des constantes matricielles.

2.1.3/ Constantes Matricielles:

Les constantes matricielles se définissent par leur valeur, ainsi la matrice:

```
| 1    2|
|       |
|3.5 7.4|
```

sera définie par:

```
[1 2; 3.5 7.4]
```

et la matrice

```
|1+3*%i   -0.3|
|              |
|7.29      %i |
```

sera définie par:

```
[1+3*%i -0.3; 7.29 %i]
```

Quelques observations sont nécessaires. En premier lieu, on peut utiliser à la place de la paire de caractères [] la paire <>. En plus, des espaces ou des virgules séparent des éléments d'une ligne, tandis que le point-virgule sépare des lignes. Pour finir, il est impératif ne pas laisser des espaces dans les expressions contenues dans des crochets.

```
1  + 3 * %i
```

serait interprété comme la matrice |1 3%i|.

La matrice vide est représentée par [] (Voir page 10).

2.1.4/ Chaînes de Caractères:

Les constantes chaînes de caractères se représentent comme la chaîne entourée d'apostrophes ou de guillemets (ces deux caractères étant équivalents en BLAISE). L'apostrophe ou le guillemet se représentent par deux apostrophes (ou guillemets) successifs. Il n'y a pas de chaîne "vide". Exemple de chaînes de caractères:

```
'entier'
"Maître Corbeau..."
```

2.1.5/ Commentaires:

Deux "slashs" (//) consécutifs dans une ligne indiquent que le reste de la ligne est ignoré. En particulier, ils sont employés pour définir des lignes de commentaire. Voir aussi getf (page 21).

2.2/ TYPE ET NOM DES VARIABLES

A défaut de donner une caractérisation formelle et rigoureuse des types de données employés dans BLAISE, nous adopterons plutôt un point de vue descriptif et fonctionnel. Vues par l'utilisateur les données et variables employées sont en principe des chaînes de caractères, des données et des variables numériques. Ces dernières sont, en général, considérées dans BLAISE comme étant réelles ou complexes, scalaires ou matricielles. Il n'y a pas dans le langage BLAISE de spécification de type; le type d'une variable est défini par les affectations dont elle résulte:

```
a = 1
```

définit "a" comme un scalaire réel et l'assignation:

```
a = 1 + %i
```

définit "a" comme un scalaire complexe. Le type d'une variable peut changer en fonction des affectations à lesquelles elle est soumise. Au niveau de l'utilisateur il n'y a pas, en général, de différences de traitement entre réels et complexes, les différences éventuelles seront mentionnées dans ce guide.

Le nom des variables est formé d'une lettre suivie d'un nombre quelconque de lettres ou nombres, mais seuls les quatre premiers caractères sont pris en compte, le premier caractère du nom peut être le caractère special % (variables prédéfinies).

Pour toutes les variables et constantes numériques, on obtient la transposée-conjuguée, en leur adjoignant le caractère apostrophe. Ainsi x' est la transposée-conjuguée de x.

2.2.1/ Commande d'assignation:

La commande d'assignation crée une variable dans le système et l'assigne une valeur. La forme générale est:

```
nom_de_variable = expression
```

ou "expression" est ou bien une constante ou une combinaison de constantes et variables obtenue en se servant des opérateurs et fonctions que nous décrirons tout au long de ce GUIDE. Par exemple:

```
a = [1 2 3; 4 5 6]
```

crée la variable a et lui donne comme valeur la matrice:

```
| 1  2  3 |
|         |
| 4  5  6 |
```

les instructions:

```
a = [1 2 3;
     4 5 6]
```

ou, encore:

```
a = [
1 2 3;
4 5 6
  ]
```

produiraient le même résultat.

La variable a est conservée dans le système et peut alors être utilisée, ou réutilisée à tout moment.

On peut écrire la commande d'assignation sans la variable de gauche, et sans le signe =. Alors une variable nommée ans ("answer") est créée automatiquement pour représenter le résultat.

2.2.2/ Compléments sur les constantes et les variables:

Maintenant que nous avons la commande d'assignation nous ajouterons quelques éléments de plus. Les matrices peuvent être introduites dans le système en ligne ou en colonne. Par exemple:

```
a = [-1.3000 0.8 %pi]
```

donne

```
|-1.3000   0.8   3.14159|
```

comme valeur de la variable a. Tandis que

```
a = [-1.3000 ;  0.8 ;  3.14159]
```

donnera

```
| -1.3000 |
|         |
|  0.8    |
|         |
| 3.14159 |
```

comme valeur de a.

Les matrices peuvent être définies par blocs. Par exemple,

```
x = [a , b ; c , d]
```

où a, b, c et d sont quatre matrices, déjà définies dans le système. Si les tailles de ces matrices ne sont pas cohérentes un message d'erreur est donné.

On peut aussi désigner un élément d'une matrice en mettant son indice entre parenthèses, par exemple a(1,1), ou de cette façon faire des assignations partielles comme a(3,7) = x.

L'expression a = [] crée une matrice vide, qui existe dans le système. Il est alors possible de faire:

```
a = [ ]
a = [a, eye(2)]
```

qui produira,

$$a = \begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix}$$

ou, encore

```
a = []
for b = [1, 2, 3, 4, 5] a = [a, f(b)]
```

La commande d'assignation permet "d'étendre" une matrice. Ainsi, si nous avons dans le système une matrice mat de dimensions (3,4), et nous faisons mat(10,10) = 8, nous obtiendrons une matrice 10 par 10, dont les éléments non définis explicitement dans le système sont mis à zéro, les autres conservant leur valeur.

De plus, des vecteurs d'indices peuvent être utilisés pour désigner une sous-matrice d'une matrice donnée. Ainsi, si x et v sont des vecteurs, alors x(v) est le vecteur

$$[x(v(1)) , x(v(2)) , \ldots , x(v(n))]$$

Un vecteur peut aussi être défini par une "boucle implicite" représentée par deux points:

$$x = m : i : n$$

est le vecteur

$$[m , m + i , \ldots , m + k * i]$$

où k est le plus grand (petit) entier tel que m + k * i est plus petit (grand) que a si i est positif (négatif).

Le même symbole ":" peut être employé pour désigner des lignes ou de colonnes d'une matrice, ainsi a(:, j) est la j-ième colonne de a, et a(j:k, :) est la matrice formée des lignes j à k de la matrice a et de toutes ses colonnes.

Remarque: les indices peuvent ne pas être entiers, seule leur partie entière est prise en compte.

Un dernier mot sur les constantes et les variables. Les macro-instructions définies par l'utilisateur sont aussi des variables. Comme cette dernière affirmation est un peu obscure, on peut ici dire au moins ceci: si mec est une macro-instruction, la commande d'assignation a = mec est valable, et a comme conséquence de définir a comme une macro-instruction copie de mec. On pourrait aussi avoir des opérations qui agissent sur les macro-instructions, mais cela n'est pas encore réalisé dans la présente version de BLAISE. Notons cependant qu'une (ou plusieurs) macro-instruction peut être utilisée comme variable d'entrée (ou de sortie) d'une autre macro-instruction et donc être executée à l'intérieur de cette dernière.

Par exemple, soit <y> = f(x) une macro-instruction définissant le vecteur y en fonction du vecteur x, et <j> = j(x) la macro-instruction "dérivée de f" calculant la matrice jacobienne j de

f en fonction de x. L'algorithme de Newton (résolution de f(x) = 0) s'écrira simplement en BLAISE par:

```
//<x> = newton(x, n, f, j)
  for k= 1 : n, x = x - j(x) \ f(x); end
```

2.2.3/ Opérations élémentaires:

Nous finirons cette première approche du système BLAISE en énumérant les opérations élémentaires. Ces opérations travaillent actuellement, uniquement sur les matrices (réelles ou complexes). Ces opérations sont:

- l'addition a + b
- la soustraction a - b
- la multiplication a * b
- la division à gauche a \ b
- la division à droite a / b

L'interprétation et la possibilité de réalisation de ces opérations dépend du type des variables sur lesquelles on opère. Par exemple, pour les matrices on a les interprétations usuelles. La syntaxe et les règles d'évaluation des expressions sont les mêmes que celles du langage FORTRAN.

2.2.4/ Sur les multiplications:

Le sens de la multiplication dépend du contexte. En particulier, toutes les matrices peuvent être multipliées par un scalaire. Si on multiplie deux matrices entre elles, leurs tailles doivent être compatibles.

D'autres produits matriciels existent. La multiplication élément par élément se note x .* y. Le produit tensoriel de Kronecker noté x .. y (page 31).

L'expression x ** p est x à la puissance p. La variable p doit être un scalaire.

Pour une matrice hermitienne x, la fonction puissance, x ** p est calculée à partir des valeurs propres et vecteurs propres v de x. Si <v, d> = spec(x) alors

```
x ** p = v * (d **p) / v
```

Pour les arguments vectoriels, la fonction est appliquée élément par élément. La fonction puissance x ** p n'est pas implantée pour les matrices non-hermitiennes.

2.2.5/ Sur les divisions / et \:

2.2.5.1/ Backslash ou division à gauche.

x = a \ b

donne la solution du systeme linéaire a * x = b. La solution est calculée par la méthode d'élimination de Gauss. Un message est imprimé si la matrice a est presque singulière. a \ eye donne l'inverse de a.

Si a est une matrice m par n avec m < n ou m > n et b une matrice m par k alors x = a \ b est la solution aux moindres carrés du système a * x = b. Le rang numérique de a est determiné par l'algorithme qr avec pivots. Une solution x avec au plus (rang de a) composantes non nulles par colonne est calculée. Si (rang de a) < n alors la solution sera différente de pinv(a) x (voir page 37).

Avec a \ eye on calcule une inverse généralisée de a.

Si a et b ont les mêmes dimensions, alors a .\ b a pour éléments a(i,j) \ b(i,j).

2.2.5.2/ Slash ou division à droite.

x = b / a

est la solution de x * a = b. Pour les matrices b/a = (a'\b')'

Si a et b sont de mêmes dimensions, alors a ./ b a pour elements a(i,j) / b(i,j).

2.3/ COMMANDES UTILITAIRES

2.3.1/ Help, what, who et clear:

Un "help" en ligne est disponible. Pour avoir de l'information sur une commande il suffit de taper "help" suivi du nom de la commande. "help" sans paramètre, donne la liste des renseignements disponibles par "help". La liste des commandes s'obtient en faisant "what".

La commande "who" donne la liste des variables, tant prédéfinies que courantes, ainsi que dimensions de la pile interne et le nombre de mots occupées par les variables. La commande "clear" tue toutes les variables définies. La commande "clear x y z ..." tue les variables x, y, z, ... Mais, attention! "clear a, b" détruit a et imprime b.

2.3.2/ Editeur:

BLAISE dispose d'un "éditeur" permettant de modifier, et de réexecuter la dernière ligne de commande tapée. Pour y accéder, il suffit de taper ":", après le "a-vous". Les sous-commandes de l'éditeur sont:

p	imprime la ligne de commande
d	détruit la ligne de commande
a/ch/	ajoute la chaîne de caractères "ch" en fin de ligne
s/ch1/ch2/	substitue toutes les ocurrences de ch1 par ch2
q	abandonne l'éditeur et exécute la commande.

2.3.3/ Diary:

La commande diary('file') génère une copie de toute la session BLAISE sur un fichier à partir du moment ou cette commande est émise et jusqu'au moment ou la commande diary(0) est tapée. Le paramètre "file" est l'identificateur du fichier (page 17).

2.3.4/ Display:

Si x est une matrice formée d'éléments scalaires, alors disp(x) produira l'impression de la matrice de signes de la partie réelle de x. C'est à dire un tableau où à la place de chaque élément on aura le signe de sa partie réelle. Ainsi, pour les éléments dont la partie réele est positive, négative ou nulle on aura respectivement +, - ou espace.

Si x est une macro-instruction disp(x) imprime sa liste d'appel et le texte source (voir page 19).

Si x est une bibliothèque disp(x) donne la liste des macro-instructions BLAISE contenues dans x (voir page 19).

2.3.5/ Formats d'impression:

BLAISE fait toujours ses calculs en double précision, mais les résultats des opérations peuvent être visualisés avec plus ou moins de chiffres décimaux. Le format habituel est de 4 chiffres après le point décimal, sauf pour les entiers qui sont imprimés sous la forme d'un entier suivi d'un point.

La commande "long" provoque l'impression des résultats avec 14 chiffres après le point décimal. Le retour au format habituel s'obtient par la commande "short".

Si on souhaite obtenir les résultats en point décimal flottant, on peut utiliser les commandes "short e" ou "long e".

Pour chaque ligne de commande émise, un compteur accumule la quantité de lignes de résultats imprimées. Lorsque cette quantité approche une valeur limite (25 par défaut) le système interroge

l'utilisateur pour supprimer ou continuer l'impression. La commande "lines(n)" fixe cette valeur limite à n.

Disons, pour finir cette section que le point-virgule en fin de commande supprime l'impression des résultats.

2.4/ INSTRUCTIONS DE CONTROLE

2.4.1/ Symboles de comparaison:

Les symboles de comparaison, ou opérateurs logiques, sont les suivantes:

=	égalité
<	plus petit que
>	plus grand que
<=	plus petit ou égal
>=	plus grand ou égal
<>	différent de

leur usage est réservé aux expressions de contrôle des clauses de contrôle.

2.4.2/ Boucle for:

La clause "for" permet de répéter des instructions BLAISE un nombre spécifié de fois.

Syntaxe:

```
for variable = var-boucle, instruction 1, instruction 2,...., end
```

Le paramètre var-boucle exige une explication approfondie, étant donnée la similitude de la clause "for" de BLAISE par rapport à celle d'autres langages. La variable de contrôle de la boucle que nous avons nommée var-boucle peut être soit une boucle implicite, telle quelle à été décrite (page 11); ou bien une variable numérique, c'est à dire une matrice, et dans ce cas, la variable prend successivement comme valeurs les vecteurs colonne de la var-boucle. On a vu antérieurement que les boucles implicites sont des cas particuliers de variables.

Exemple 1:

```
for j = 1 : n, for k = 1 : n, a(k,j) = 1 / (k + j - 1), end, end.
```

L'instruction précédente génère la matrice de Hilbert.

Exemple 2:

```
m = [1  2+3*%i  -4.35 ; %i  -0.5  17.2 ; 0  1.4+7*%i  1]
v = []
for k = m, v=[v k'*k]
```

La notation k' signifie la transposée-conjuguée de k (page 9). Les trois instructiones antérieures nous permettent de former un vecteur ligne, dont les éléments sont les carrés de la norme l2 des colonnes de la matrice m.

2.4.3/ Boucle while:

Cette clause permet de répéter des instructions un nombre de fois indéfini:

Syntaxe:

while expr1 oplog expr2, instruction1, instruction2, ..., end

où expr1, expr2, sont deux expressions et oplog est l'un des symboles de comparaison défini en page 15.

Les instructions sont répétées tant que le test de comparaison entre les deux expressions est satisfait.

Exemple:

```
tol = 1;
while 1 + tol > 1, tol = tol / 2, end; eps = 2 * tol
```

cet exemple calcule la précision additive de l'ordinateur sur lequel on travaille.

2.4.4/ Clause if-then-else:

Cette clause permet l'exécution conditionnelle d'instructions.

Syntaxe:

if expr1 oplog expr2 then instructions else instructions, end.

Le premier groupe d'instructions est exécuté si la relation de comparaison est vraie, dans le cas contraire le second groupe d'instructions est exécuté.

Le mot-clé "then" peut être remplacé par une virgule. La partie du else (et les instructions qui le suivent), peut être omise.

2.4.5/ Remarques:

Pour les trois dernières clauses, le dernier mot-clé end peut être remplacé, en mode conversationnel, par le retour-chariot et, en mode exec ou getf, par la fin de fichier.

Les virgules qui séparent les instructions peuvent être remplacées par des points-virgule.

En mode macro (ou exec) une clause peut s'écrire sur plusieurs lignes; la fin de la clause est alors indiquée par le mot-clé "end".

2.4.6/ Clause pause:

La commande pause suspend l'exécution de la ligne de commande, au point ou elle est placée et rend la main a l'utilisateur pour insérer une nouvelle ligne. L'exécution de la ligne suspendue reprenant dès la fin d'execution de la ligne insérée. La commande pause suivie d'un texte imprime ce texte avant d'effectuer la pause. La fin du texte est indiquée par une virgule. Dans cette version l'instruction "pause" n'est pas compilable (voir page 23).

2.5/ INSTRUCTIONS D'ENTREE-SORTIE

2.5.1/ Généralités:

Les commandes dont les descriptions suivent sont spécialement concernées par la vocation modulaire et extensible de BLAISE.

2.5.2/ Paramètre file:

Les commandes exec, getf, save, load, print, write, read et diary, accèdent à des fichiers.

Le paramètre "file" peut prendre des formes différentes selon les divers systèmes d'exploitation. Sur la plupart d'eux, "file" peut être une chaîne de caractères.

Par exemple, dans save('a') ou exec('Blaise/demo.exec') la chaîne sera utilisée comme nom d'un fichier dans le système hôte. Sur tous les systèmes, file peut être un entier positif inferieur à 10 qui sera utilisé comme unité logique FORTRAN.

La commande exec('file') est équivalente à la suite d'instructions suivantes (naturellement, c'est vrai pour toutes les autres commandes qui se servent du paramètre 'file'):

```
a = 'file'
exec(a)
```

2.5.3/ Load et save:

La commande save('file') stocke toutes les variables courantes sur un fichier en binaire avec une structure particulière, contenant pour chaque variable, en plus de données le nom de la variable et sa longueur.

La commande save('file', x, y, ...) stocke les variables x, y, ... Les variables sauvées peuvent être restaurées par la commande load.

La commande load('file') charge dans BLAISE les variables contenues dans un fichier créé par la commande save.

La commande load('file', x, y, ...) charge les variables x, y, ... à partir du fichier par la chaîne 'file' crée par la commande save.

2.5.4/ Write et read:

La commande write('file', a, form), permet d'écrire le contenu de la matrice a dans le fichier défini par file selon le format FORTRAN form. Le paramètre a correspond à une variable BLAISE de type matrice réelle, tandis que form représente une chaîne de caractères definissant le format FORTRAN desiré, par exemple '(8e10.3)'. Les parenthèses sont obligatoires.

La commande x = read('file', m, n, form) lit la matrice réelle x à partir du fichier défini par file, selon le format FORTRAN représenté par form. Les paramètres m, n sont les dimensions souhaitées de la matrice x; naturellement, ces dimensions doivent être compatibles avec les données disponibles.

2.5.5/ Exec:

La commande exec('file', k) à pour effet la lecture et l'exécution des instructions BLAISE contenues dans un fichier et qui sont écrites d'après les mêmes règles que celles ecrites en conversationnel.

L'impression est controllée par le paramètre optionnel k. Si k = 0 il n'y a pas d'écho, ni promot, ni pause. C'est la valeur par défaut si la commande exec est suivie d'un point-virgule. Si k = -1 rien n'est imprimé. Si k = 1 on reçoit un écho de chaque ligne de commande. Si k = 2 le prompt <> est imprimé. Si k = 3 il y a échos et prompts, mais pas de pauses. C'est la valeur par défaut si la commande exec n'est pas suivie d'un point-virgule. Si k = 4, BLAISE pause avant chaque prompt et attend une ligne blanche pour continuer. Si k = 7 il y a pauses, prompts et échos. Ce mode est utile pour les démonstrations. La commande exec(0) fait que les commandes qui suivent sont attendues sur le terminal. La fin de fichier a le même effet. Le texte BLAISE du fichier peut lui même contenir des exec.

2.5.6/ Print:

La commande print('file', x) écrit la variable x sur le fichier dont le nom est 'file' selon le format d'impression courant de BLAISE. La commande print ne peut écrire qu'une seule variable.

2.5.7/ Bibliothèques:

La commande x = lib('file') permet d'identifier le fichier défini par 'file' comme étant une bibliothèque de macro-instructions. Le système peut alors charger automatiquement ces macro-instructions lors de leur premier appel.

Le paramètre 'file' désigne un fichier qui doit avoir été créé auparavant par la commande save. La variable x est une variable composite de type liste qui contient le nom du fichier (donné par save) et la liste des noms de macro-instructions contenues dans ce fichier. Pour visualiser le contenu de cette variable on peut se servir de disp(x).

Les règles de recherche automatique des variables fonctionnent de la façon suivante; quand on fait référence à une variable le systeme la recherche d'abord dans sa base de données et puis dans les diverses variables de type liste, dans l'ordre de leur création. Pour supprimer une bibliothèque il faut détruire la variable-liste associée.

2.5.8/ Remarques:

Les fichiers auxquels on accède en écriture (write, save, diary, ...) ne doivent pas exister au préalable (option: "status = new" du open de FORTRAN). Après chaque ordre de lecture ou d'écriture, le fichier est refermé. Par conséquent un nouvel ordre d'écriture sur le même fichier conduit soit à une erreur, soit à en ecraser le contenu selon le système hôte.

2.6/ DEFINITION DES MACRO-INSTRUCTIONS

2.6.1/ Généralités:

Comme on l'a déja dit le système BLAISE permet à l'utilisateur de définir les extensions qu'il souhaite en écrivant des petits programmes en langage BLAISE. Une fois définis, ces programmes ou macro-instructions s'utilisent de façon aussi souple que les fonctions propres du système BLAISE. Par contre, tandis que les fonctions BLAISE sont des primitives du langage, lesquelles ne peuvent être soumises a des opérations, les macro-instructions sont des variables BLAISE. Elles peuvent être affectées par la commande d'assignation: si mec est une macro-intruction et que nous faisons:

f = mec

alors b = mec(u) et b = f(u) donneront la même valeur de b. Tandis que mec = 3, fera disparaître notre macro-instruction du système et la variable mec vaudra bien 3.

De façon analogue, disp appliquée à une fonction ne nous donnera rien, tandis qu'apoliquée à une macro-instruction disp(macro) imprime sa liste d'appel et le texte source.

Les macro-instructions peuvent être passées en argument dans la liste d'appel d'une autre macro-instruction, mais cela n'est pas possible pour une fonction.

Il existe deux commandes ("deff" et "getf") pour définir les macro-instructions.

2.6.2/ Commande deff:

Cette commande permet de définir directement dans BLAISE une nouvelle macro-instruction.

Syntaxe:

```
deff('<u, v, ...> = macro(x, y, ...)', 'texte')
```

où

- macro est le nom donné à la macro-instruction nouvellement définie.

- <u, v, ...> est la liste des variables de sortie qui peut être omise, s'il n'y a pas des variables de sortie.

- texte est une suite d'instructions BLAISE séparées indifféremment par des virgules, des points_virgules ou des points de suspension, la longueur de texte est limitée a environ 1024 caractères. Pour des macro-instructions plus comolexes il faut utiliser la commande getf (page 21).

Exemple 1:

```
deff("[a] = dup(b)", "a = 2 * b")
```

est la définition de la macro-instruction qui va multiplier par 2 l'argument.

Exemple 2:

```
deff("[ro,teta]" = pol(x, y)", "ro = sqrt (x**2+y**2), teta=b/a")
```

Maintenant qu'on commence à avoir une certaine intimité avec les macro-instructions on se permettra de les appeler macros.

2.6.3/ Commande argn:

Permet de connaître dans une macro le nombre de variables d'entrée et de sortie avec le quel elle a été appelée:

Syntaxe:

```
<rhs, lhs> = argn(0)
```

ou

```
<lhs> = argn(0)
```

où:

- rhs: nombre de variables de sortie demandée,
- lhs: nombre de variables d'entrée fournies,

au moment de l'appel de la macro.

2.6.4/ Commande return:

Cette commande émise à l'interieur d'une macro ou dans un exec en termine l'exécution et provoque le retour au terminal ou à la fonction appelante.

2.6.5/ Commande getf:

Cette commande permet de charger dans Blaise une ou plusieurs macros. Une fois chargées les macros sont utilisables dans Blaise comme des commandes Blaise standard. La syntaxe est getf('file') où file est l'identificateur du fichier sur lequel est editée la macro (ou les macros). La première ligne du fichier doit être de la forme suivante:

```
//<s1, s2, ..., slhs> = nom_macro(e1, e2,..., erhs)
```

où les ei sont les variables d'entrée et les si les variables de sortie. Les lignes suivantes du fichier contiennent une suite d'instructions BLAISE. Toutes les variables apparaissant dans le texte sont locales sauf les variables d'entrée et sortie. On peut separer les différentes lignes par des virgules, des points-virgules ou des points de suspension. Chaque fonction doit être terminée par une ligne //end ou par la fin de fichier. Les clauses doivent se terminer par le mot "end" (ou par la fin de fichier).

Exemple de fichier:

```
//<g,d>=fact(a,tol)
//factorisation de plein rang
<u, s, v> = svd(a), s = diag(s), v = v',
k = 1,
while s(k) > tol, k = k+1, end,
```

```
rang = k - 1, s = s(1: rang), s = sqrt(s), s = diag(s),
g = u(: , 1: rang) * s, d = s * v(1: rang, : )
```

2.6.6/ Variables et récursivité:

2.6.6.1/ Variables:

Dans une macro, les instructions peuvent utiliser:

- les variables d'entrée de la macro.

- les variables internes définies dans les instructions précédentes de la macro.

- les variables globales préexistantes dans le système avant l'appel de la macro.

Les variables globales sont accessibles en "lecture", mais elles sont protegées en "écriture"; si la macro affecte une variable dont le nom est identique à celui d'une variable globale, une variable interne du même nom est creée. Dans la suite de la macro, toute référence à ce nom de variable désignera la variable interne.

Les variables d'entrée sont considérées comme des variables internes, et peuvent être modifiées sans modifier les variables d'appel.

A la fin de l'exécution de la macro, les variables internes sont détruites.

Exemple 1:

```
<x> = toto(a)
y = a * a' + eye
x = y ** n + x
n = 4
```

est valide. Son appel suppose que les variables n et x soient définies. Si l'on a:

x = 10 et n = 2

l'instruction u = toto(3) affectera à u la valeur 110:

((3 * 3 + 1) ** 2 + 10)

quant aux variables x et n elles vaudront toujours après l'execution 10 et 2, respectivament.

Exemple 2: les macros

```
<x> = t1(a)          et          <x> = t2(a)
b(3) = 1                         b
```

```
x = b * b' + a                    b(3) = 1
                                  x = b * b' + a
```

ne fournissent pas le même résultat, y = t1(a) affectera à y la valeur 1 + a, par contre y = t2(a) conduira à une erreur si b n'est pas définie. Si b est définie, soit b = [1 2 4], cette instruction affectera à y la valeur 6 + a. La deuxième ligne de t2 est équivalent a b = b, la variable b qui est à droite est la variable globale, et celle de gauche crée une variable locale du même nom dont les contenus sont une copie de ceux de la variable globale.

2.6.6.2/ Recursivité:

- Une macro peut appeler n'importe quelle macro déja définie et en particulier elle-même.

Exemple:

```
<n> = fact(m)
if m > 1,then n = m * fact(m - 1), else n = 1, end.
```

calcule m!. La profondeur de la récursion est limitée et dépend du contexte dans lequel est utilisée la macro.

2.6.7/ Commande comp:

La commande comp(macro) génère une nouvelle variable macro qui est la version compilée de la macro donnée. Cette version compilée s'utilise de façon totalement identique aux macros interprétées, en particulier les macros compilées peuvent appeler et être appelées par des macros compilées ou non. La compilation des macros permet un gain de temps important, qui est de l'ordre de 2 a 3 par rapport a l'interprétation. Après compilation il ne sera plus possible de connaître le "source" de la macro.

2.6.8/ Commande fort:

La commande fort permet d'établir rapidement l'interface vers un sous-programme FORTRAN avec BLAISE.

Syntaxe:

```
<y1, ..., yk> = fort(ident, x1, px1, 'tx1', ..., xn, pxn, 'txn',
    'sort', <ny1, my1>, py1, 'ty1', ..., <nyl, myl>, pyl, 'tyl')
```

où la signification des différents paramètres est la suivante:

- ident: nom du programme FORTRAN appelé (facilité non portable, disponible momentanément seulement en Multics) ou nombre entier qui est transmis a la routine d'interface interf.

- x1, ..., xn: variables d'entrée du sous-programme appelé.

- px1, ..., pxn: positions respectives de ces variables dans la liste d'appel du sous-programme appelé (ident).

- tx1, ..., txn: types FORTRAN respectifs de ces variables. Ils sont codifiés de la façon suivante:

r	réel
i	entier
d	double precision
e	external (seulement pour Multics)

Tout autre type est interdit.

- 'sort': mot-clé reservé pour séparer les variables d'entrée de celles de sortie.

- <ny1, my1>, ...: taille, c'est à dire nombre de lignes et de colonnes des variables de sortie.

- py1, ...: positions des variables de sortie (éventuellement confondues avec px).

- 'ty1', ...: types FORTRAN des variables de sortie.

Les k premières variables de sortie sont mises dans y1, ..., yk.

La routine interf, dont le sorce est accessible à l'utilisateur effectue l'appel au programme FORTRAN. Cette routine reçoit les arguments d'entrée xi dans l'ordre spécifié par les pxi et dans les types txi.

L'utilisateur doit écrire dans la routine interf, si par exemple il veur appeler le programme toto comportant 4 variables d'entrée:

```
call toto (x1, x2, x3, x4)
return
```

Il n'est donc pas nécessaire de dimensionner ni de déclarer les variables xi. L'entier ident est passé par le common/inter/ident. En fonction de cet entier l'utilisateur gère l'appel aux routines qu'il désire interfacer, par exemple grace à un goto calculé.

2.6.9/ Commande user:

La commande user appelle le programme d'interface (voir le chapitre INTERFACES BLAISE-FORTRAN, page 47) MATUSR, et lui transmet les valeurs des paramètres d'appel. L'utilisateur peut ainsi interfacer des programmes FORTRAN avec BLAISE.

Le dernier paramètre sert à désigner le programme FORTRAN interfacé par MATUSR que l'on veut appeler.

3/_FONCTIONS_BLAISE

3.1/_CALCUL_NUMERIQUE

3.1.1/_Fonction_abs:

La fonction abs(x) donne la valeur absolue des elements de x (ou le module dans le cas complexe).

3.1.2/_Fonction_bdiag:

La fonction bdiag effectue la bloc-diagonalisation de la matrice réelle a.

Syntaxe:

[ab, x, bs] = bdiag(a, {rmax})

où:

- bs: donne la structure des blocs (tailles respectives des différents blocs).
- x: donne le changement de base: x \ a * x est la matrice bloc-diagonale ab.
- rmax: est un paramètre facultatif: sa valeur par défaut est la norme l1 de a. rmax contrôle le conditionnement de x (cf. Bavely et Stewart).

Les commandes ab = bdiag(...) ou <ab, x> = bdiag(...) sont aussi valides.

3.1.3/_Fonction_black:

La fonction black produit le diagramme de black de la fraction rationnelle H = n/d, c'est à dire la courbe log(abs(H(jw))) par rapport à arg(H(jw)). Cette commande graphique n'est à executer qu'à partir d'un terminal graphique.

Syntaxe:

black(n, d, wmin, wmax, n_pt)

où:

- n et d: vecteurs qui représentent les coefficients des polynômes numérateurs et dénominateurs rangés par puissances croissantes,
- wmin et wmax: pulsations minimales et maximales,
- n_pt: quantité maximum de points à utiliser dans la discrétisation.

3.1.4/ Fonction bode:

La fonction bode produit les diagrammes de bode de la fraction rationnelle f = n/d , c'est à dire les courbes mod(f(iw)) et arg(f(iw)). Attention: cette commande n'est à executer qu'à partir d'un terminal graphique.

Syntaxe:

bode(n, d, wmin, wmax, n_pt)

- n et d: vecteurs représentant les coefficients des polynômes numérateurs et dénominateurs rangés par puissances croissantes.
- wmin et wmax: pulsations minimales et maximales.
- n_pt: nombre maximum de points à utiliser dans la discrétisation.

3.1.5/ Fonction chol:

La fonction chol réalise la factorisation de Cholesky d'une matrice. Si x est une matrice, alors chol(x) utilise seulement la diagonale et la partie triangulaire supérieure de x. La partie triangulaire inférieure est supposée la transposée (conjuguée) de la partie supérieure. Si x est définie-positive, alors r = chol(x) est une matrice triangulaire supérieure telle que r'*r = x . Si x n'est pas définie-positive on reçoit un message d'erreur.

3.1.6/ Fonction cond:

Conditionnement en norme l2. cond(x) est le rapport de la plus grande à la plus petite valeur singulière de x (voir svd, page 43).

3.1.7/ Fonction conj:

La fonction conj(x) donne le complexe-conjugué de x.

3.1.8/ Fonction cont:

La commande cont donne la forme controllable d'une paire (a,b).

Syntaxe:

<n, [u]> = cont(a, b, [tol])

où,

- n: dimension du sous-espace de controllabilité

- tol: tolérance dans le calcul de rang (qr)

- u: changement de base orthogonal qui met la paire (a, b) sous forme canonique. Les n premières colonnes de u donnent une base orthogonale du sous-espace de controllabilité. Si v =u(:, 1:n), alors v'*a*v et v'*b donnent la partie controllable de la paire (a, b).

3.1.9/ Fonction cos:

La fonction cos(x) donne le cosinus de x.

Pour une matrice hermitienne x, la fonction cos est calculee à partir des valeurs propres et vecteurs propres v de x. Si <v, d> = spec(x) alors cos(x) = v * cos(d) / v. Pour les arguments vectoriels, la fonction est appliquée élément par élément. La fonction cos n'est pas implantée pour les matrices non-hermitiennes.

3.1.10/ Fonction det:

La fonction det(x) donne le déterminant de la matrice rélle ou complexe x.

3.1.11/ Fonction diag:

Si v est un vecteur (ligne ou colonne) à n composantes, diag(v, k) est une matrice carrée d'ordre n + abs(k) avec les éléments de v sur la k-ième diagonale. Le cas k = 0 correspond à la diagonale principale, tandis que k > 0 correspond aux sur-diagonales et k < 0 aux sous-diagonales. La fonction avec un seul paramètre, diag(v), est la matrice diagonale diag(v, 0).

Exemple:

```
diag(-m: m) + diag(ones(2 * m, 1), 1) + diag(ones(2 * m, 1), -1)
```

donne une matrice tridiagonale d'ordre 2 * m + 1.

Si x est une matrice, diag(x, k) est un vecteur colonne formé des éléments de la k-ième diagonale de x. diag(x) est la diagonale principale de x. diag(diag(x)) est une matrice diagonale.

3.1.12/ Fonction ent:

La fonction ent(x) donne la partie entière de x.

3.1.13/ Fonction exp:

La fonction exp(x) est l'exponentielle de x. Pour une matrice hermitienne x, la fonction exponentielle est calculee à partir des valeurs propres et vecteurs propres v de x. Si <v, d> = spec(x) alors exp(x) = v * exp(d) / v. Pour les arguments vectoriels, la fonction est appliquée élément par élément. Pour les matrices quelconques exp(x) est calculée par approximants de Padé sur la forme bloc-diagonale de x (voir bdiag, page 25).

3.1.14/ Fonction freq:

La fonction freq donne la réponse fréquentielle d'un système linéaire representé en variables d'état.

Syntaxe:

x = freq(a, b, c, f)

où:

- (a, b, c): matrices réelles de tailles (n, n), (n, p) et (m, n), respectivement,

-f: vecteur (complexe des frequences de taille (1, t) ou (t, 1),

-x: matrice de taille (m, p*t) telle que

x(:, k * p : (k+1) * p) = c * inv(f(k) * eye - a) * b

3.1.15/ Fonction frk:

La fonction frk(n) engendre la matrice de Franck:

||a(i, j||

où:

a(i, j) = n - max(i, j) si i est plus petit ou égal à j + 1 et a(i, j) = 0 dans le cas contraire.

3.1.16/ Fonction gsch:

Cette fonction donne la forme de Schur généralisée d'un faisceau s * e - a.

Syntaxe:

[as, es] = gschur(a, e)

ou bien,

<as, es, q, z> = gschur(a, e)

où:

- as: matrice quasi-triangulaire supérieure,

- es: matrice triangulaire supérieure.

- q et z: transformations orthogonales à gauche (q) et à droite (z) qui mettent le faisceau sous la forme de Schur as, es (algorithme qz), c'est à dire:

$$q * (s * e - a) * z = s * es - as.$$

3.1.17/ Fonction gspec:

La fonction gspec donne le spectre généralisé d'un faisceau s * e -a, ou e et a sont des matrices réelles, c'est à dire les racines de l'équation det(s * e - a) = 0.

Syntaxe:

```
[al, be] = gspec(a, e)
```

ou

```
<al, be, z> = gspec(a, e)
```

si be(i) = 0, alors la i-ème valeur propre est infinie. Pour obtenir les valeurs propres il faut faire al ./ be. (Dans le cas e = eye on obtient le spectre de a quoiqu'il soit plus efficace d'utiliser la commande spec(a) (page 42)). La deuxième forme

```
<al, be,  z> = gspec(a, e)
```

donne en plus les vecteurs propres à droite dans la matrice z.

3.1.18/ Fonction hess:

Cette commande donne la forme de Hessenberg d'une matrice. Cette forme d'une matrice est telle que les éléments placés sous la première sous-diagonale sont nuls, elle est déduite par un changement de base.

Syntaxe:

```
h = hess(a)
```

ou

```
[p, h] = hess(a)
```

où:
- h: matrice de Hessenberg telle que a = p * h * p',

- p: matrice unitaire de changements de base orthogonal.

Si la matrice est symétrique ou hermitienne la forme est tridiagonale.

3.1.19/ Fonction hilb:

Cette fonction donne l'inverse de la matrice de Hilbert. La matrice de Hilbert est ||1 / (i + j - 1)||, laquelle, par construction est très mal conditionnée. Le resultat est exact pour n inferieur a environ 15 selon les machines.

3.1.20/ Fonction imag:

La fonction imag(x) donne la partie imaginaire de x.

3.1.21/ Fonction impl:

La fonction impl s'applique à la résolution de systèmes différentiels implicites:

$$a(t, y) * dy/dx = g(t, y)$$

Syntaxe:

y = impl({'type'), y0, ydot0, t0, t1, {atol, {rtol}}, res, adda, {jac})

où:

- y0: condition initiale (vecteur donné)
- ydot0: derivée à l'instant initial
- t0: instant initial
- t1: vecteur des instants où est calculée la solution
- res: macro BLAISE du type <r>=res(t,y,ydoy) calculant r=g(t,y)-a(t,y)*ydot
- adda: macro BLAISE du type <x>=adda(t,y,p) calculant x=a(t,y)+p

Paramètres optionnels(voir ode, page 33):

- 'type': 'adams' ou 'raid'
- jac: macro BLAISE définissant le jacobien de la fonction res de type <j>=jac(t,y,ydot)
- atol, rtol: tolérance (voir ode, page 33)

Les fonctions res, adda et jac peuvent aussi etre des routines FORTRAN: on donne alors leur nom entre guillemets (Multics) ou on les interface par les routines fres, fadd, et fj2 (voir le source de ces routines pour les listes d'appel). Les types mixtes sont autorisés, mais res et adda doivent être du même type.

3.1.22/ Fonction inv:

La fonction inv(x) donne l'inverse de la matrice x. Un message apparait si la matrice est mal conditionnée. (Voir aussi pinv pour la pseudo-inverse (page 37) (inversion par svd, matrice non nécéssairement carrée).)

3.1.23/ Fonction kron:

Le produit tensoriel de kronecker de x et y est noté kron(x, y) on le note aussi x .*. y. Le résultat est une matrice formée en prenant tous les produits possibles entre les elements de x et ceux de y.

Exemple: si x est une matrice d'ordre 2 par 3, alors x .*. y vaut

[x(1, 1)*y x(1, 2)*y x(1, 3)*y x(2, 1)*y x(2, 2)*y x(2, 3)*y]

BLAISE connait aussi des quotients tensoriels de Kronecker notés: x ./. y et x .\. y, lesquels sont définis comme plus haut en remplaçant la multiplication par la division à droite ou à gauche.

3.1.24/ Fonction log:

La fonction log(x) donne le logarithme naturel de x. Le résultat est complexe si x n'est pas positif ou définie-positive.

Pour une matrice hermitienne x, la fonction log est calculée à partir des valeurs propres et vecteurs propres v de x. Si <v, d> = spec(x) alors log(x) = v * log(d) / v. Pour les arguments vectoriels, la fonction est appliquée élément par élément. La fonction log n'est pas implantée pour les matrices non-hermitiennes.

3.1.25/ Fonction lu:

La fonction lu donne les facteurs de l'élimination de Gauss.

Syntaxe:

[l, u] = lu(a)

où:

- u: matrice triangulaire supérieure,
- l: matrice triangulaire inférieure à une permutation des colonnes près.

Naturellement, on aura a = l * u.

3.1.26/ Fonction lyap:

Etant donnée l'équation de Lyapounov "continue",

a' * x + x * a = c , (c symétrique)

la commande

lyap(a, c, 'cont')

donne sa solution. Par contre, la solution de l'équation de Liapounov "discrète",

a' * x * a - x = c , (c symétrique)

est donnée par la commande

lyap(a, c, 'disc').

On suppose que toutes les matrices intervenantes sont réelles.

3.1.27/ Fonction magic:

La fonction magic(n) construit un carré magique d'ordre n, c'est à dire une matrice d'ordre n à éléments entiers plus petits que n**2, et telle que les sommes de ses éléments sur les lignes et les sommes de ses éléments sur les colonnes sont égales.

3.1.28/ Fonctions maxi et mini:

La fonction maxi(v) donne le plus grand élément du vecteur ou de la matrice v. Attention! ces fonctions n'admettent qu'un seul argument en entrée. Avec deux paramètres de sortie

<m, k> =maxi(v)

donnera, en plus, l'indice du plus grand élément dans le vecteur, ou les indices de ligne et de colonne dans la matrice.

De façon complètement analogue, la fonction mini(v) donne l'élément minimum du vecteur ou de la matrice v, et en continuant le parallélisme,

<m, k> = mini(v)

donne en plus l'indice du plus petit élément dans le vecteur, ou les indices de ligne et de colonne dans la matrice.

3.1.29/ Fonction norm:

La fonction norm de BLAISE calcule les normes les plus usuelles de vecteurs et des matrices. Dans la suite, on va énoncer toutes ces possibilités.

Normes pour les Matrices:

- norm(x) ou norm(x, 2) est la plus grande valeur singulière de x.
- norm(x,1) est la norme l1 de x.
- norm(x, 'inf') est la norme "infini" de x.
- norm(x, 'fro') est la norme de Frobenius, c'est à dire

```
sqrt(sum(diag(x' * x)))
```

Normes pour les Vecteurs:

- norm(v, p) est la norme lp, c'est à dire

```
(sum(v(i) ** p)) ** (1 / p)
```

- norm(v) est la norme l2, c'est à dire c'est égal à norm(v, 2).
- norm(v, 'inf') est la norme infini de v, c'est à dire max(abs(v(i))).

3.1.30/ Fonction nyq:

La fonction nyq(n,d,wmin,wmax,npt) produit le diagramme de Nyquist de la fraction rationnelle n/d, c'est à dire la courbe imag(n(iw) / d(iw)) en fonction de real(n(iw) / d(iw)) pour w = wmin, ..., wmax. Les vecteurs n et d représentent les coefficients des polynômes numérateurs et dénominateurs rangés par puissances décroissantes, wmin et wmax sont les pulsations minimales et maximales, npt le nombre maximum de points à utiliser dans la discrétisation. Attention: cette commande ne fonctionne que sur un terminal graphique.

3.1.31/ Fonction ode:

La fonction ode s'emploie pour la résolution d'équations différentielles ordinaires. Ainsi,

```
yt = ode(['type'], y0, t0, t, [rtol, [atol]], ydot, [jac])
```

integre l'équation différentielle

```
dy/dt = f(t, y)
```

où:

- y0 = y(t0) est la condition initiale (vecteur colonne),

- t0 = instant initial,

- t: vecteur (il peut être quelconque) des instants où est calculé yt, la i-ème colonne de yt contenant y(t(i)),

- ydot: référence externe en BLAISE ou en FORTRAN, c'est à dire, que c'est ou bien le nom d'une macro BLAISE définissant le second membre f(t, y). Dans ce cas ydot(t, y) doit être un vecteur colonne. Ou bien (facilité reservée au système MULTICS) une variable BLAISE de type "chaine de caractères" (c'est à dire entre guillemets ou apostrophes) donnant le nom d'un sous-programme FORTRAN définissant f(t,y), dont la liste d'appel sera, par exemple:

```
fex(ny, t, y, ydot)
```

avec des notations évidentes.

Lorsque la chaîne de caractères est un entier (entre guillemets ou apostrophes), le programme d'intégration appelle le sous-programme FORTRAN fydot(ny, t, y, ydot) et cet entier y est passe au common /kydot/. L'utilisaterur doit alors faire exécuter sa propore routine FORTRAN à l'interieur du sous-programme fydot, par exemple par l'instruction

```
if (kydot .eq. 12) call fex(ny, t, y, ydot)
```

Exemple 1: ode(y0, t0, t1, 'fex') fait appel au sous-programme FORTRAN fex (non portable).

Exemple 2: ode(y0, t0, t1, '12') fait appel au sous-programme du système fydot et transmet l'entier 12 au common /kydot/.

Variables optionelles:

- rtol, atol: erreurs (relative et absolue) estimées. L'erreur sur y(i) est estimée par

```
rtol(i) * abs(y(i)) + atol(i)
```

donc, le test d'arrêt est positif si dans chaque composante l'erreur rélative est inférieure à rtol ou l'erreur absolue inférieure à atol.

Ces paramètres peuvent être scalaires lorsque leurs composantes sont égales. Les valeurs par défaut sont:

```
rtol = 1.d-5   et   atol = 1.d-7.
```

- 'type': Une des chaînes de caractères 'adams' ou 'raide' selon que le système est non-raide (méthode d'Adams) ou raide

(méthode bdf). Seule la première lettre est testée. Si le type n'est pas precisé le choix se fait automatiquement.

- jac: référence externe en BLAISE ou en FORTRAN, c'est à dire, ou bien le nom de la macro définissant le jacobien du système, ou bien le nom d'un sous-programme FORTRAN. Sont valables les mêmes conventions que pour ydot. La liste d'appel pour un programme FORTRAN sera

jac(ny, t, y , m, n, jac, njac)

(m et n ne sont pas référencés, njac est le nombre de lignes du tableau jac). Lorsque la chaîne de caractères est un entier (entre guillemets ou apostrophes), le sous-programme FORTRAN nommé

fjac(ny, t, y, m, n, jac, njac)

et cet entier y est passe au common /kjac/.

3.1.32/ Fonction opti:

Sur BLAISE il existe une certaine variété de traitements des problèmes d'optimisation, le lecteur intéressé pourra voir le sujet plus extensivement traité dans l'Appendice B. Ici nous nous bornerons à une exposition succinte. BLAISE permet de résoudre des problèmes de minimisation sans contraintes ou avec des contraintes de borne au moyen de la fonction opti.

Nous indiquerons avec accolades {} les paramètres optionnels.

Syntaxe:

<f, {x, {g, {tr}}}> = optim(nclas, nap, x, simul, epsg, {options}),

avec {options} = {{binf, bsup}, {"in"}, {tr}, {itmax}}

où:

- f: valeur du critère,

- x: variable à optimiser,

- g: gradient de f en x,

- tr: tableau de travail permettant le démarrage à chaud dans les méthodes de quasi-Newton. Lors du premier appel il est initialisé par optim. Ensuite, on peut l'utiliser en entrée d'optim pour accélérer l'optimisation. A n'utiliser que pour nclas = 10.

- nclas: définit le type du critère et des contraintes et l'algorithme utilisé pour résoudre le problème. Sont implantés les cas:

- nclas = 10 critère régulier, sans contraintes (BFGS)

- nclas = 20 critère régulier, contraintes de borne (BFGS + projection)

- nclas = 21 critère régulier, contraintes de borne (BFGS à mémoire limitée plus projection). Il convient si la dimension de x est grande).

- nap: nombre maximum d'appels de simul,

La fonction simul permet de calculer f et g. C'est, soit une macro BLAISE de la forme

<f, g, ind> = simul(x, ind)

Si ind = 2 (respectivement 3, 4) on calcule f (respectivement <g, f> et g). Si ind = 1, rien n'est calculé (il sert aux impressions). En sortie, ind < 0 signifie que f n'est pas définie en x et ind = 0 implique l'arrêt de l'optimisation.

- le nom entre guillemets ou apostrophes d'une subroutine FORTRAN dont la liste d'appel est de la forme:

(ind, n, x, f, g, izs, rzs, dzs)

Les tableaux de travail izs, rsz, dzs sont reservés à l'utilisateur; ind est comme ci-dessus, sauf pour l'initialisation eventuelle de paramètres: voir ci-dessous.

- un entier i. Si sim est le nom de la subroutine de calcul de f et g, il faut alors écrire dans la subroutine FORTRAN d'interface foptim (fournie dans BLAISE) l'ordre:

if(koot .eq. i) call sim(ind, n, x, f, g, izs, rzs, dzs)

- epsg qui est un test d'arrêt sur la norme du gradient (projeté)

- binf, bsup sont des bornes inférieures et supérieures (nclas = 20 ou nclas = 21)

-"in", chaîne de caractères de longueur 2,elle commande l'initialisation de paramètres par simul (ne sert que si simul est une subroutine FORTRAN). Les deux premiers appels de simul se font alors avec ind = 10 et puis ind = 11. Si ind = 10, simul doit affecter les dimensions de izs, rzs et dzs dans le common:

common/nird/ nizs, nrzs, ndzs

Si ind = 11, simul doit initializer les variables izs, rzs et dzs.

-itmax nombre maximum d'iterations de l'algorithme.

3.1.33/ Fonction orth:

Celle-ci est une fonction d'orthogonalisation.

```
q = orth(x)
```

est une matrice a colonnes orthogonales (q' * q = identité) dont les colonnes engendrent le même sous-espace que celles de x.

3.1.34/ Fonction pinv:

Cette fonction calcule la pseudo-inverse d'une matrice, x = pinv(a) donne une matrice x de meme taille que a', telle que soient vérifiées les relations:

```
a * x * a = a
```

et

```
x * a * x = x
```

avec a*x et x*a hermitiennes. Le calcul passe par la svd de a et les valeurs singulières inférieures à un seuil donné sont considerées comme nulles. Le seuil par défaut est

```
norm(size(a),'inf') * norm(a) * eps
```

Ce seuil peut être modifié par la commande x = pinv(a, seuil) (voir rank, page 39).

3.1.35/ Fonction poly:

La fonction poly donne le polynôme caractéristique dans le cas de matrices.

Si a est une matrice d'ordre n par n, alors poly(a) est un vecteur-colonne dont les n + 1 éléments sont les coefficients du polynôme caractéristique: det(lambda * eye - a).

Si v est un vecteur, alors poly(v) est un vecteur dont les éléments sont les coefficients du polynôme ayant les éléments de v comme racines. Pour les vecteurs, roots (page 41) et poly sont des fonctions inverses l'une de l'autre (a l'ordre près et aux erreurs d'arrondi près). Pour illustration notons que roots(poly(1: 20)) engendre l'exemple bien connu de Wilkinson.

3.1.36/ Fonction ppol:

La fonction ppol place les pôles d'un système linéaire mono-entrée. Plus précisement, la fonction

k = ppol(a, b, pol),

détermine le vecteur-ligne k tel que le spectre de la matrice

a - b * k

soit celui défini par le vecteur pol.

La signification des différentes variables qui participent à la fonction est:

- a: matrice d'ordre n x n, réelle
- b: vecteur d'ordre n x 1 réel
- pol: vecteur d'ordre n.

Bien entendu, la paire (a, b) doit être controllable, c'est à dire:

<b, a * b, ..., a**(n - 1) * b>

de rang plein, ou encore, en utilisant la commande cont, on doit avoir n = cont(a, b).

3.1.37/ Fonction prod:

Etant donnée x, prod(x) est le produit de tous les éléments de x. Disons, à titre d'exemple que prod(1: n) est la factorielle de n.

3.1.38/ Fonction qr:

La fonction qr donne la factorisation du même nom.

En particulier, <q, r> = qr(a) donne une matrice triangulaire supérieure r de même taille que x et une matrice unitaire q telle que x = q * r. Et <q, r, e> = qr(x) donne une matrice de permutation e, une matrice triangulaire supérieure et une matrice unitaire q telles que x *e =q * r.

3.1.32/ Fonction rand:

La fonction rand est un générateur de nombres aléatoires.

Les différentes modalités d'appel sont les suivantes:

- rand(n): matrice aléatoire d'ordre n.

- rand(m, n): matrice aléatoire d'ordre m par n.

- rand(a): étant donné un objet a, est un objet aléatoire de même taille que a.

- rand: appelé sans arguments, est un scalaire aléatoire qui est modifié à chaque appel. Les tirages étant distribués selon une loi uniforme sur (0,1).

- rand('normal'): bascule les tirages suivants sur une loi normale (moyenne 0, variance 1).

- rand('uniform'): rétablit la loi uniforme.

- rand('seed'): donne la valeur courante de l'argument du générateur.

- rand('seed',n): remet l'argument du générateur à n, la valeur initiale de n au premier appel à Blaise est 0, rand('seed', 0) reinitialise le générateur.

3.1.40/ Fonction rank:

La fonction rank donne le rang numérique d'une matrice.

La commande k = rank(a) donne le nombre de valeurs singulières de x qui sont supérieures à

norm(size(x), 'inf') * norm(x) *eps

Tandis que k = rank(x, tol) est le nombre de valeurs singulières de x qui sont supérieures a tol.

3.1.41/ Fonction rat:

La fonction rat fait des approximations rationnelles.

Ainsi, <n, d> = rat(x, [prec]) approche les coefficients de la matrice x par une fraction continue de la forme :

a/b = d1 + 1/(d2 + 1/(d3 + ... + 1/dk))

de telle sorte que

norm(x - n ./ d, 1) < prec

Si le paramètre prec est omis sa valeur par défaut est %eps. La forme rat(x, [prec]) donne, simplement, l'approximation rationnelle de x à prec près.

3.1.42/ Fonction rcond:

La fonction rcond(x) est un estimateur de l'inverse du conditionnement de x en norme l1. Si x est bien conditionnée, rcond(x) est proche de 1. Si x est mal conditionnée, rcond(x) est proche de 0.

La forme <r, z> = rcond(x) met r à la valeur rcond(x) et donne, aussi un vecteur z tel que

norm(a * z, 1) = r * norm(a, 1) * norm(z, 1)

ainsi si rcond est petit, z approche un vecteur du noyau.

3.1.43/ Fonction real:

La fonction real(x) donne la partie réelle de x.

3.1.44/ Fonction ricc:

La fonction ricc(a, b, c, 'cont') donne une solution x de l'équation de Riccati "continue":

a' * x + x * a - x * b * x + c = 0

Les matrices a, b, c doivent être réelles. Les matrices b et c sont supposées définies non-négatives et la paire (a, g) stabilisable avec g * g' factorisation de plein rang de b, la paire (h, a) est détectable avec h * h' factorisation de plein rang de c. La solution est alors unique et définie non-négative.

La forme ricc(f, g, h, 'disc') donne une solution de l'équation de Riccati discrète:

x = f'*x*f-f'*x*g1*((g2+g1'*x*g1)-1)*g1'*x*f+h

f est supposée inversible et g =g1 * (g2 - 1) * g1'. On suppose (f, g1) stabilisable et (c, f) détectable avec c' * c factorisation de plein rang de h. La solution est alors unique et définie non-négative.

3.1.45/ Fonction roots:

La fonction roots(c) calcule les racines du polynôme dont les coefficients sont les éléments du vecteur c. Si c a n+1 composantes, le polynôme s'écrit

```
c(1) * x**n + ... + c(n) * x + c(n + 1)
```

Voir aussi poly (page 37).

3.1.46/ Fonction round:

La fonction round(x) arrondit les éléments de x à l'entier le plus proche.

3.1.47/ Fonction rref:

La fonction rref(a) donne la forme ligne échelon réduite d'une matrice rectangulaire. La forme rref(a, b) est équivalente à rref(<a, b>); avec b = eye(a) on obtient la matrice x telle que x * a est sous forme réduite.

3.1.48/ Fonction schur:

La fonction schur donne la décomposition de Schur d'une matrice. Ainsi <u, t> = schur(x) produit une matrice triangulaire supérieure t, dont la diagonale est constituée des valeurs propres de x, et une matrice unitaire u telle que

```
x = u * t  *  u'
```

La commande abregée schur(x) donne t. Si x est réelle alors <u, t> donne la forme de Schur réelle de x avec des blocs de dimensions 1*1 ou 2*2 sur la diagonale.

Pour une matrice réelle, une autre forme de la commande est:

```
<u, d> = schur(x, 'disc')
```

ou

```
<u, d> = schur(x, 'cont')
```

On obtient, alors la forme de Schur "ordonnée", c'est à dire, les d premières colonnes de la matrice orthogonale u forment une base du sous-espace spectral (invariant) associé aux valeurs propres de module inferieur a 1 (pour 'disc'), ou à parties réelles négatives (pour 'cont'). Si t = u(:, 1 : d) la matrice u' * a * u est donc stable.

3.1.49/ Fonction sin:

La fonction sin donne le sinus de x.

Pour une matrice hermitienne x, la fonction sin est calculee à partir des valeurs propres et vecteurs propres v de x. Si <v, d> = spec(x) alors sin(x) = v * sin(d) / v. Pour les arguments vectoriels, la fonction est appliquée élément par élément. La fonction sin n'est pas implantée pour les matrices non-hermitiennes.

3.1.50/ Fonction size:

Si x est une matrice d'ordre m par n, alors size(x) est le vecteur [m, n]. Avec des assignations multiples on peut écrire <m, n> = size(x). Si x est une variable de type caractère, size(x) est le nombre de caractères de la variable.

3.1.51/ Fonction sort:

La fonction sort(v), avec v réelle, donne le vecteur résultant du tri de v par ordre décroissant. La forme <s, k> = sort(v) donne de plus les indices des éléments de s en v.

3.1.52/ Fonction spec:

La fonction spec(a) donne le vecteur-colonne des valeurs propres de la matrice carrée a. La fonction bdiag permet d'avoir une décomposition en valeur et vecteur propre (si la matrice est diagonalisable); il faut alors choisir emax grand.

3.1.53/ Fonction sqrt:

La fonction sqrt(x) donne la racine carrée de x.

Pour une matrice hermitienne x, la fonction sqrt est calculee à partir des valeurs propres et vecteurs propres v de x. Si <v, d> = spec(x) alors sqrt(x) = v * sqrt(d) / v. Pour les arguments vectoriels, la fonction est appliquée élément par élément. La fonction sqrt n'est pas implantée pour les matrices non-hermitiennes. Le résultat est complexe si x est négatif ou n'est pas défini non-négatif.

3.1.54/ Fonction stab:

Sous-espace stable à droite d'un faisceau réel s *e - a. La forme stab(a, e, 'cont') donne une base orthogonale de vecteurs propres généralisés correspondant aux valeurs propres a partie réelle négative. La forme stab(a,e,'disc') accomplit la même action que la forme antérieure pour les valeurs propres dans le cercle unité.

3.1.55/ Fonction sum:

La fonction sum(x) donne la somme de tous les éléments de x. En particulier, sum(diag(x)) est la trace de la matrice x.

3.1.56/ Fonction sva:

La fonction sva fait l'approximation par valeurs singulières.

La forme <u, s, v> = sva(a, k), où k est un entier supérieur ou égal à 1, donne, par troncature de la svd (page 43) de a a ses k plus grandes valeurs singulières, la meilleure approximation l2 de la matrice a par une matrice b de rang k, à savoir b = u * s * v'. La norme l2 de a - b est alors égale à la plus grande valeur singulière negligée.

La forme <u, s, v> = sva(a, tol), où tol est un réel inférieur à 1, donne une approximation l2 de a par troncature des valeurs singulières de a aux valeurs superieures a tol. L'approximant b = u * s * v' est tel que la norme l2 de a - b est au plus égale a tol. Lorsque tol n'est pas precisé, la valeur par défaut prise est égale à max(m, n) * eps * norm où (m, n) est la taille de a et norm sa norme l2.

3.1.57/ Fonction svd:

La fonction svd produit la décomposition en valeurs singulières. La forme <u, s, v> = svd(x) donne une matrice diagonale s, de même dimension que x, et à éléments diagonaux non-négatifs classés par ordre décroissant, et deux matrices unitaires u et v telles que x = u *s *v'. Par contre, svd(x) ne donne que le vecteur des valeurs singulières. Et la forme <u, s, v> = svd(x, 0) produit la decomposition "economy size". Si x est d'ordre m par n avec m > n, alors seules les n premières colonnes de u sont calculées et s est n par n. Pour finir, la forme <u, s, v, r> = svd(x) donne en plus le rang r de x(voir rank, page 39).

3.1.58/ Fonction sylv:

La fonction sylv(a, b, c, 'cont') donne la solution x de l'équation de Sylvester "continue" a * x + x * b = c, où a, b et c sont matrices réelles de tailles compatibles.

3.1.59/ Fonction tan:

La fonction tan(x) donne la tangente des éléments de x. Pour une matrice hermitienne x, la fonction tan est calculee à partir des valeurs propres et vecteurs propres v de x. Si <v, d> = spec(x) alors tan(x) = v * tan(d) / v. Pour les arguments vectoriels, la fonction est appliquée élément par élément. La fonction tan n'est pas implantée pour les matrices non-hermitiennes.

3.1.60/ Fonction tril:

La fonction tril donne le triangle inférieur d'une matrice. Elle a deux formes; tril(x) est la partie triangulaire inférieure de x. Tandis que tril(x,k) est constituée des éléments sur, et au dessous de la k-ième diagonale de x. La valeur k = 0 corresponde à la diagonale principale, k > 0 est au dessus de la diagonale principale et k < 0 au dessous de la diagonale principale.

3.1.61/ Fonction triu:

La fonction triu donne le triangle superieur des matrices. Il est possible de l'appeler sous différentes formes; triu(x) est la partie triangulaire superieure de x. Tandis que triu(x, k) est constituée des éléments sur et au dessus de la k-ième diagonale de x. La valeur k = 0 répresente la diagonale principale, k>0 est au dessus de la diagonale principale et k<0 au dessous.

3.1.62/ Fonction tzer:

La fonction <al, be> = tzer(a, b, c, d) calcule les zéros de transmission d'un système linéaire representé en variables d'état par un quadruplet (a, b, c, d). Les zéros sont donnés par al ./ be. Les matrices a, b, c, d doivent être réelles et de tailles compatibles.

3.2/ GRAPHIQUE

Le système BLAISE possède des capacités graphiques interactives assez developpées. La commande de base est

plot(x, y)

qui a pour effet de tracer la courbe donnée sous la forme d'une succession de points (x(i), y(i)) dans le plan. x et y sont ici deux vecteurs.

Un exemple intéressant est le suivant:

```
t = 0:50; plot(t .* cos(t), t .* sin(t))
```

L'environnement graphique est entièrement défini par une variable BLAISE nommée dess (vecteur à 72 composantes) qui prend des valeurs par défaut lorsque la commande plot est activée. En configurant à sa guise la variable dess, l'utilisateur définit un environnement graphique adéquat qu'il peut, bien entendu, sauver et restaurer.

La signification des éléments de la variable dess est donnée dans l'Appendice A.

La commande

```
plot({x}, y, {'legx', {'legy', {'leg_dessin'}}})
```

où les accolades signifient, comme habituellement, que les paramètres enfermés sont optionnels. Cette fonction produit le dessin de y par rapport à x, avec eventuellement des legendes sur les axes et/ou la courbe. Si le paramètre x est omis il est pris par défaut égal au vecteur 1:size(y). Les vecteurs x et y sont réels et legx, legy et leg_dessin sont des chaînes de caractères.

La commande

```
plot(x, y, z, {legx, {legy, {legz}}})
```

produit le dessin de la courbe paramètrée x(z) par rapport a y(z), les valeurs du paramètre z étant inscrites sur la courbe.

A l'aide de macros qui changent des variables dess spécifiques on peut tracer des figures variées. Par exemple, la macro suivante trace les pôles (*) et zéros (o) d'une fraction rationnelle n / d:

```
//poles(n,d)
dess=getd(1)
nr=roots(n)
dr=roots(d)
ni=imag(nr);nr=real(nr)
di=imag(dr);dr=real(dr)
max=maxi(<nr;dr;1;ni;di>)
min=mini(<nr;dr;-1;ni;di>)
dess(<2,7:10,55>)=<0;min;min;max;max;9>//definition du cadre
//trace des 2 nuages de points
plot(nr,ni)
dess(54:55)=<1;2>
plot(dr,di)
//trace des axes reel et imaginaire
dess(55:56)=<3;0>//trait pointille
plot(<min max>,<0 0>)
plot(<0 0>,<min max>)
dess(55)=1//trait plein
cercle(<0 0>,1)
```

```
//trace du cercle unite
//cercle(orig,r)
t=0:0.05:2.1*%pi
plot(orig(1)*ones(t)+r*sin(t),orig(2)*ones(t)+r*cos(t))
```

D'autres commandes graphiques telles que Bode, Black, Nyquist sont également disponibles.

4/ INTERFACES BLAISE-FORTRAN

4.1/ LA BASE DE DONNEES

4.1.1/ Structure FORTRAN:

La base de donn'ees de BLAISE est constituée de quatre tableaux FORTRAN:

IDSTK: tableau entier de taille IDSTK(4, LSIZE). IDSTK(1:4, K) contient les 4 caractères significatifs du nom de la variable numéro K, représentés par leur code BLAISE.

LSTK: vecteur d'entiers de taille LSIZE. LSTK(K) contient l'adresse de début de la variable K dans la pile STK, alors que LSTK(K+1)-1 est l'adresse du dernier mot de STK de cette variable.

STK: vecteur de double precision de taille VSIZE contenant les définitions de toutes les variables reconnues par BLAISE et de la zone de travail.

ISTK: vecteur entier en equivalence avec STK (il occupe la même zone de la mémoire).

Quatre pointeurs complètent cette description:

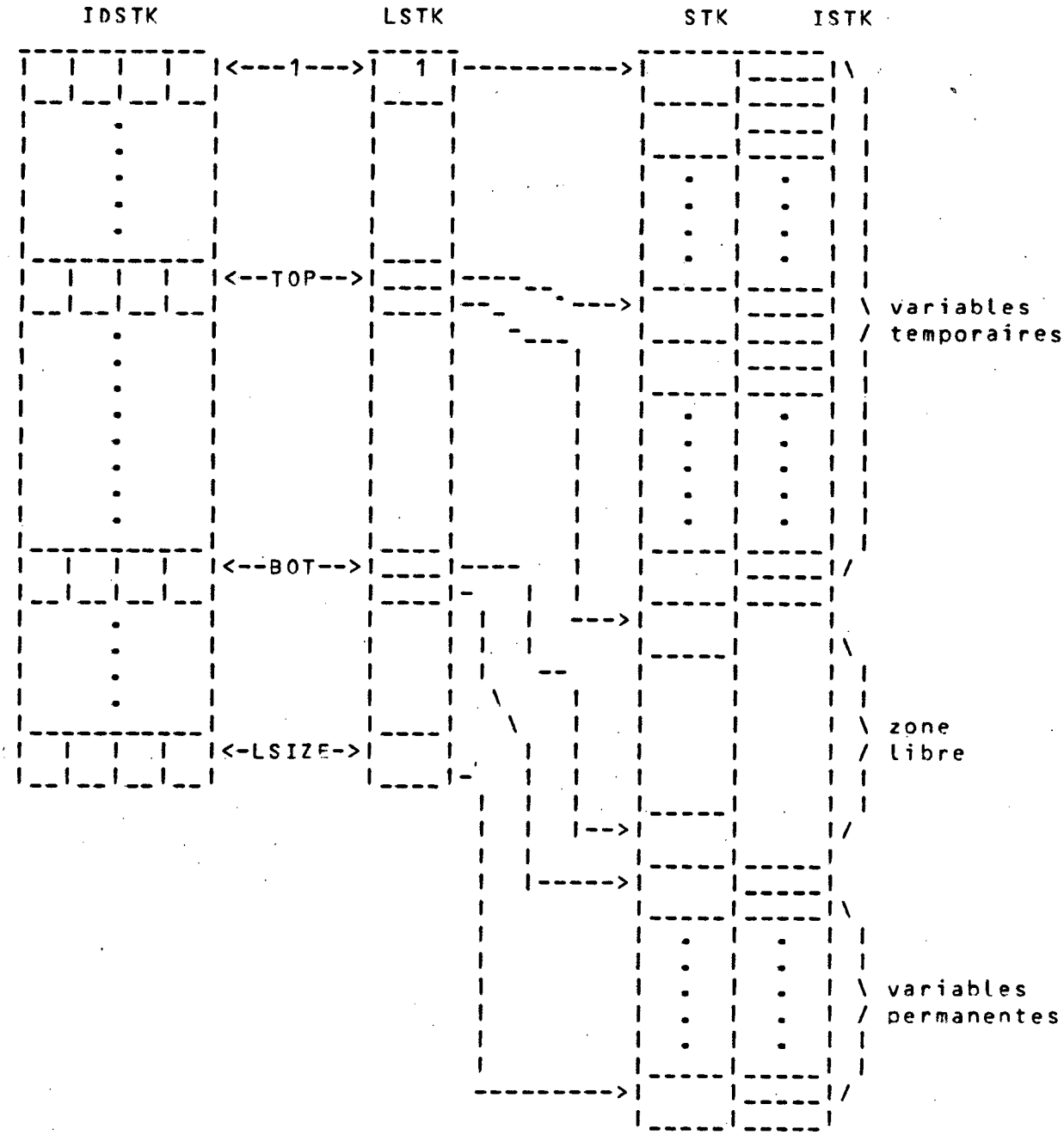
LSIZE: dimensionnement des tableaux IDSTK et LSTK. LSIZE défini le nombre maximum de variables (permanentes et temporaires) pouvant exister à un instant donné dans le système.

VSIZE: dimensionnement de la pile STK. VSIZE défini le volume maximum compté en mots double précision pouvant être occupé par les variables permanentes, les variables temporaires et la zone de travail.

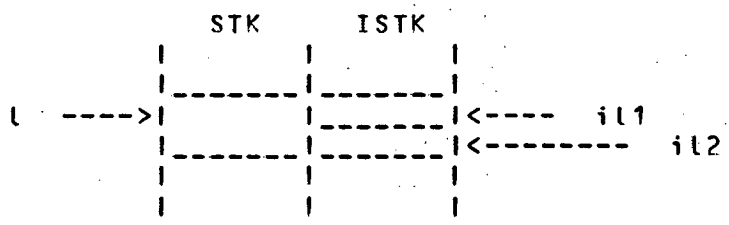
VSIZE et LSIZE sont des constantes qui ne peuvent être modifiées qu'en changeant leur affectation, et le dimensionnements de tableaux IDSTK, LSTK, STK dans le sous-programme MATLAB.

TOP: pointeur relatif aux tableaux LSTK et IDSTK: les variables de numéro 1 à TOP sont des variables temporaires (argument de fonction ou de macro, évaluation de sous-expressions, ...). LSTK(TOP+1) est la première adresse libre de la pile STK.

BOT: les variables de numéros BOT à LSIZE-1 sont les variables permanentes (resultent d'une affectation: a=expr...). LSTK(BOT)-1 est le dernière adresse libre de la pile STK. La relation TOP+1<BOT doit toujours être verifiée (écrasement). Le graphique suivant donne une image de la description précédente.



REMARQUE: Sur la plupart des machines un "mot double precision" équivaut à deux mots entiers. Pour rendre le système portable, la conversion d'adresse entre STK et ISTK est effectuée par l'intermédiaire de la fonction addr. L'adaptation au système hôte s'effectue en modifiant cette fonction.



on a les relations:

```
il1 = adr(l, 0)
l = adr(il1, 1)
adr(il2, 1) = l + 1
```

la base de données est transmise aux différents sous-programmes par les commons étiquetés:

```
COMMON /STACK/ STK
COMMON /VSTK/ IDSTK, LSTK, VSIZE, LSIZE, BOT, TOP, LEPS
```

LEPS est l'adresse de STK où est rangée la valeur de la précision machine b**(1-t).

4.1.2/ Codage des différents types de variable:

Soit k le numéro de la variable considerée et il=adr(LSTK(k),0), il pointe sur le premier mot de la pile ISTK relatif à cette variable. ISTK(il) défini le type de la variable.

a) type matrice scalaire:

ISTK(il) est égal à 1.

ISTK(il + 1) contient le nombre de lignes m de la matrice.
ISTK(il + 2) contient le nombre de colonnes n.
ISTK(il + 3) indique si les coefficients de la matrice sont réels (ISTK(il + 3) = 0) ou complexes (ISTK(il + 3) = 1).

Soit l1 = adr (il + 4, 1), alors

STK(l1:l1+m*n-1) contient les parties réelles des coefficients de la matrice, l'élément (i,j) est rangé dans STK(l1+(i-1)+(j-1)*m).

Si STK(il + 3) est égal à 1, alors

STK(l1+m*n:l1+2*m*n-1) contient les parties imaginaires des coefficients, la partie imaginaire de l'élément (i, j) est rangée dans STK(l1+m*n+(i-1)+(j-1)*m).

b) chaînes de caractères

ISTK(il) est égal à 10:

ISTK(il + 1) contient le nombre n de caractères de la chaîne.
ISTK(il + 2) contient les codes BLAISE des caractères (voir page 3).

c) macros:

ISTK(il) est égal à 11. Dans ce cas, la suite de la variable est decomposée en 3 champs; le premier décrit les paramètres de sortie de la macro, le second les paramètres d'entrée et le troisième les instructions.

Si ils = il + 1:

ISTK(ils) contient le nombre n de paramètres de sortie.
ISTK(ils+1:ils+4*n) contient les noms de variables de sortie en code BLAISE sur quatre caractères.

Soit ile=ils+4*n+1, alors

ISTK(ile) contient le nombre n de paramètres d'entrée.
ISTK(ile+1:ile+4*m) contient les noms des variables d'entrée en code BLAISE et en quatre caractères.

Soit ilt = ile + 4*n + 1, alors:

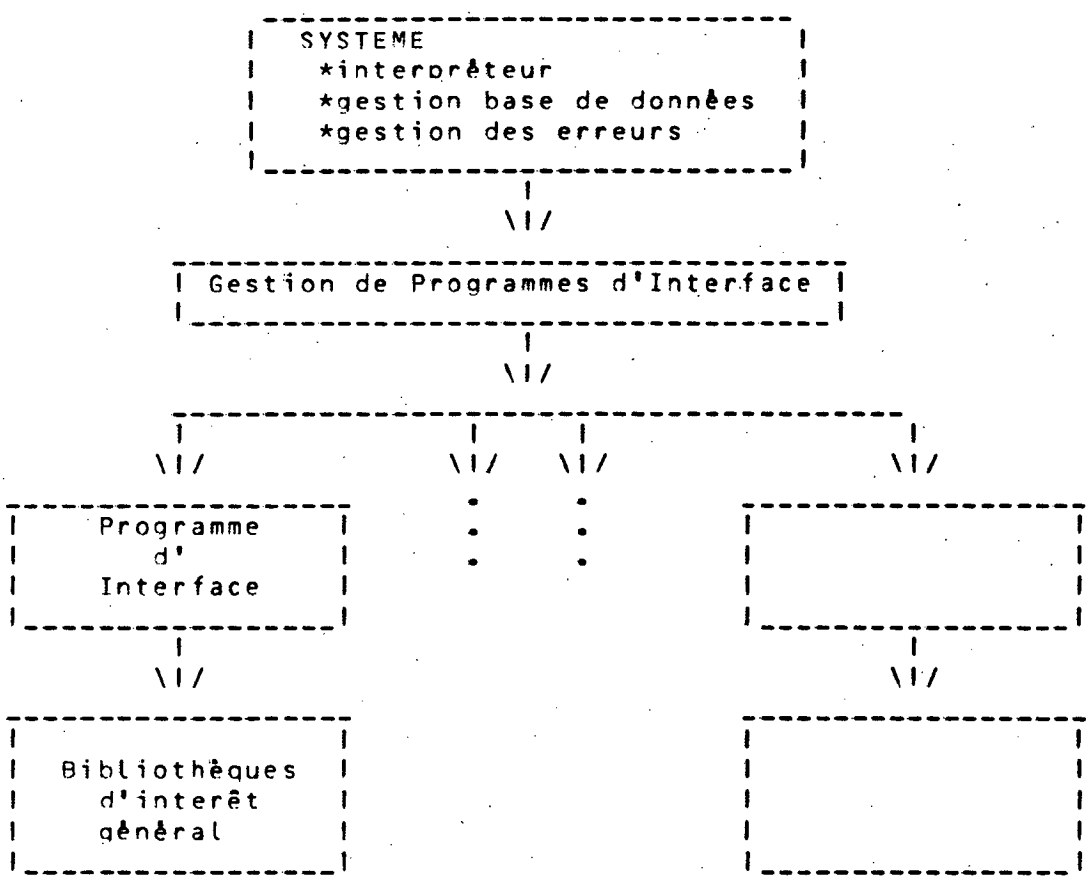
ISTK(ilt) contient la longueur l, en nombre de caractères du texte de la macro.
ISTK(ilt+1:ilt+l) le texte de la macro en code BLAISE.

Pour les macros compilées (voir page 23) ISTK(ilt+1:ilt+l) contient une suite d'entiers definissant le code opératoire.

4.2/ INTERFACES SYSTEME-BLAISE

4.2.1/ Généralités:

Rappelons tout d'abord la structure de BLAISE:



4.2.2/ Gestion des interfaces:

Le sous-programme "funs" manipule deux tables internes initialisées par data. Une table (funn(4, funl)) contient les noms des fonctions reconnues par BLAISE, codées sur 4 caractères en code BLAISE (funl est le nombre de fonctions). Une autre table (funp(funl)) définit pour chacune de ces fonctions deux entiers fun et fin, representés par l'entier 100*fun+fin, où:

fun: désigne le programme d'interface qui implémente la fonction.

fin: désigne la fonction à l'interieur du programme d'interface.

l'essentiel du travail du sous-programme funs est, étant donné un nom de fonction, de vérifier si ce nom appartient à la table funn et si oui de déterminer les valeurs correspondantes de fun et de fin. L'appel de l'interface est alors effectué par le système.

Pour rajouter une nouvelle fonction il faut donc rajouter son nom (en code BLAISE) dans la table funn, rajouter son code dans la table funp et mettre à jour la valeur de funl.

4.2.3/ Sous-programme d'interface:

A l'appel d'une fonction BLAISE le système appelle le sous-programme d'interface correspondant, en ayant au préalable défini les variables fin, lhs et rhs du common /com/ et configuré la base de données: common /stack/ et /vstk/.

4.2.3.1/ Variables lhs et rhs:

Ces variables indiquent le nombre d'arguments à gauche (lhs) et à droite (rhs) qui ont été utilisés lors de l'appel de la fonction.

Exemple:

[x, y] = toto (a, b, c) donnerait lhs = 2 et rhs = 3.

L'interface doit tester si ces valeurs correspondent aux valeurs admissibles pour les fonctions. La possibilité d'avoir des listes de paramètres de longueur variable permet de définir des variantes d'une même fonction.

Si le nombre de paramètres d'appel est incompatible, le sous-programme d'interface appele le sous-programme de gestion d'erreurs (error) avec le code 41 (lhs) ou 42 (rhs) et rend la main.

4.2.3.2/ Base de données et programme d'interface:

A l'entrée du programme d'interface les variables correspondants aux indices top-rhs+1 à top contiennent les valeurs de paramètres d'appel de la fonction. Dans l'exemple précédent le paramètre a correspondrait à la variable numéro top-2, b à top-1 et c à top. Remarquons au passage que top-2 ne vaut pas forcément 1: si l'on exécute la ligne prog(u,toto(a,b,c) à l'entrée de l'interface implantant la fonction toto la "partie haute" de la base de données contiendra les valeurs des paramètres u, a, b, c rangés dans cet ordre. Il ne faut donc, en aucun cas, modifier la partie de la base de données relative aux variables 1 à top-rhs.

4.2.3.3/ Rôle de l'interface:

Dès l'entrée, l'interface teste la valeur de fin et déroute le programme sur la partie du code qui réalise la fonction. C'est cette partie que nous détaillons par la suite:

i) analyse des variables d'entrée, pour extraire de la base de données les adresses et les caractéristiques des paramètres d'appel et tester leur validité. Si l'une d'entre elles n'est pas valide il appelle le sous-programme error avec le code 44.

Exemple:

Pour une matrice on peut écrire:

```
il = adr(lst( ), 0)
itv = istk(il)
m = istk(il + 1)
n = istk(il + 2)
it = istk(il + 3)
l = adr (il + 4, 1)
```

itv doit dans ce cas être égal à 1, m et n sont les nombres de lignes et de colonnes de la matrice, it le type (réel ou complexe) de ses coefficients; l est l'adresse du coefficient (1,1) de la matrice, les coefficients suivants sont en séquence, colonne par colonne.

La connaissance du nombre de paramètres d'entrée et de sortie et des caractéristiques de la séquence de ces paramètres peut permettre de distinguer les différentes options d'une même fonction.

ii) définition des implantations dans la pile stk ou istk des tableaux (FORTRAN) nécessaires au calcul; tableaux de travail ou tableaux de résultats et tester si la zone de travail (comprise entre lstk(top+1) et lstk(bot-1) est suffisamment grande. Sinon appel au sous-programme error après avoir affecté la variable FORTRAN err à last-lstk(bot) ou last est égal à l'adresse du dernier mot de stk necessaire, et return.

iii) appel du sous-programme, ou de la séquence de sous-programmes réalisant le calcul de la fonction. En cas d'erreur dans cette partie, appel du sous-programme error avec un code quelconque, mais plus grand que 900, en ayant au préalable affecté à la variable FORTRAN buf la chaîne de caractères constituant le message d'erreur (plus petit que 80 caractères).

Exemple:

Si l'appel est [x, y] = toto(a, b, c) la séquence des variables de sortie dans la base de données devra être x puis y.

Les implantations des differentes variables de sortie doivent être consécutives dans la pile: si k désigne une variable de la base de données lstk(k+1) pointe sur le premier mot libre de stk après la variable k et sur le premier mot rélatif à la variable k+1, en particulier lstk(top-rhs+lhs) doit être affecté à la première adresse libre après les variables de sortie. S'il n'y a pas de variables de sortie il faut affecter la valeur 0 à lhs. Enfin, positionner top à top-rhs+lhs et rendre la main.

4.2.3.4/_Remarque:

Cette structure n'est que indicative, elle peut être modifiée à volonté, aux conditions suivantes:

- ne pas modifier les zones de la base de données rélatives aux variables 1 à top-1 et bot à lsize.

- configurer correctement les variables de sortie.

- affecter top à top-rhs+lhs.

4.2.3.5/_Exemple:

```
      subroutine matexe
c
c     evaluate utility functions
c
      double precision stk(1)
      integer idstk(4,128),lstk(128),istk(1),adr
      integer vsize,lsize,bot,top
      integer alfl,case
      integer ids(4,64),pstk(64),rstk(64),psize,pt,niv,macr,paus
      integer ddt,err,fmt,lct(4),lin(1024),lpt(6),hio,rio,wio,rte,
      integer sym,syn(4),char,fin,fun,lhs,rhs,ran(2),comp(2)
      character alfa(53)*1,alfb(53)*1,buf*256
      common /stack/ stk
      common /vstk/ idstk,lstk,vsize,lsize,bot,top,leps
      common /alfs/ alfl,case
      common /recu/ ids,pstk,rstk,psize,pt,niv,macr,paus
      common /iop/ ddt,err,fmt,lct,lin,lpt,hio,rio,wio,rte,wte
      common /com/ sym,syn,char,fin,fun,lhs,rhs,ran,comp
      common /char/ alfa,alfb,buf
      equivalence (istk(1),stk(1))
      double precision pythag
c
      if (ddt .eq. 1) write(wte,1000) fin
 1000 format(1x,'matexe',i4)
c
c     functions/fin
c     abs  sort
c      1    2
c
      if(top-rhs+lhs+1.ge.bot) call error(18)
      if(err.gt.0) return
c
      il=adr(lstk(top),0)
      m=istk(il+1)
      n=istk(il+2)
      it=istk(il+3)
      l=adr(il+4,1)
      mn=m*n
c
      goto (10,100) fin
c
c abs
```

```
c
   10 l1=l-1
      if(it.eq.1) goto 12
c
c       cas reel
      do 11 i=1,mn
   11 stk(l1+i)=dabs(stk(l1+i))
      goto 999
c
c       cas complexe
   12 do 13 i=1,mn
   13 stk(l1+i)=pythag(stk(l1+i),stk(l1+i+mn))
      istk(il+3)=0
      lstk(top+1)=l+mn
      goto 999
c
c sort
c
  100 if(it.ne.0) call error(44)
      if(err.gt.0) return
c
c     test de dimensionnement
      lw=adr(lstk(top+1),0)
      err=adr(lw+n,1)-lstk(bot)
      if(err.gt.0) call error(44)
      if(err.gt.0) return
c
c     appel du sous programme
      call dsort(stk(l),mn,istk(lw))
      if(lhs.eq.1) goto 999
c
c     vecteur des index
      top=top+1
      il=lw
      l1=adr(il+4,1)+mn
      l2=lw+mn
      err=l1-lstk(bot)
      if(err.gt.0) call error(9)
      if(err.gt.0) return
      lstk(top+1)=l1
      do 101 i=1,mn
      stk(l1-i)=dble(float(istk(l2-i)))
  101 continue
      istk(il)=1
      istk(il+1)=m
      istk(il+2)=n
      istk(il+3)=0
      goto 999
c
  999 return
      end
```

4.2.4/ Interface matusr:

Créer une nouvelle interface nécessite de modifier quelques sous-programmes du système (matlab, bydot, bjac, bres, boptim, badd, ...). Matusr est une interface vide prévue pour que l'utilisateur puisse introduire facilement de nouvelles commandes. Deux modes d'utilisation sont possibles:

a) Soit comme une interface classique. Matusr correspond à fun=14. Pour introduire une nouvelle commande, il faut alors:

- écrire le corps de l'interface.

- modifier le sous-programme funs pour introduire le nouveau nom, le code associé sera alors 14xx.

b) Soit à travers la commande user. Dans ce cas il n'est pas besoin de modifier le sous-programme funs, mais il n'y a qu'un seul nom reconnu de BLAISE (user). On distingue alors les differentes fonctions interfacées par matusr par la valeur d'un paramètre, le plus simple étant d'ajouter à la liste d'appel un dernier paramètre spécifiant le numéro de l'interface dans matusr.

Exemple:

```
user(a, b, c, 3)
```

conduira à appeler matusr avec une dernière variable contenant la valeur "3". Il suffit alors dans matusr de faire:

```
il = adr(lstk(top), 0)
l = adr(il + 4, 1)
fin = idint(stk(l))
top = top - 1
rhs = rhs -1
```

pour retrouver la situation habituelle (ce qui est fait dans le sous-programme fourni).

4.3/ BLAISE A PARTIR DE FORTRAN.

Nous reste à traiter dans cette section l'utilisation de BLAISE à partir d'un programme FORTRAN. Nous croyons que l'exemple suivant d'un programme FORTRAN éclaircit suffisamment la question.

```
c       exemple d'appel de blaise par fortran
c       le fichier exemple.blaise est le fichier de commandes Blaise
c       a faire executer par Blaise.
c          contenu du fichier exemple.blaise:
c//<x>=exemple(a,b)
c//calcul de x solution de a*x=b   a et b matrice de tailles convenables
c//supposees ici reelles.
c  x=a\b
c      ici a,b et x sont des variables Blaise. Il faut donc que
c      le programme fortran affecte des valeurs
```

```
c       aux matrices a et b (a(1,1)=....), puis leur affecte les noms Bl
c  a et b (data ida..) et enfin transmette valeurs et noms a Blaise (cal
c         programme fortran appelant blaise:
       dimension a(2,2),b(2),x(2)
       double precision a,b,x
       integer ida(4),idb(4),idx(4)
       data ida/10,36,36,36/
       data idb/11,36,36,36/
       data idx/33,36,36,36/
c         on definit a et b
c
       a(1,1)=1.d0
       a(2,1)=2.d0
       a(1,2)=3.d0
       a(2,2)=4.d0
       b(1)=1.d0
       b(2)=0.d0
c         on passe les matrices a et b a Blaise en leur donnant
c         comme noms Blaise a et b respectivement...
c
c      appel muet a Blaise pour initialisations
       call matlab(-1)
c
c     variables fortran transmises en variables Blaise.
       call matz(a,a,2,2,2,ida,1)
       call matz(b,b,2,2,1,idb,1)
c      on appelle Blaise avec le numero d'unite logique 10
c       apres avoir affecte l'unite logique 10 au fichier de commandes
c       Blaise exemple.blaise
       open(unit=10,file='>udd>Blaise>o>exemple.blaise',
      1           form='formatted')
       call matlab(10)
c      on recupere le resultat x en transformant la variable
c      Blaise x en la variable fortran x.
       call matz(x,x,2,m,n,idx,0)
       write(6,100) x(1),x(2)
  100  format(2x,f10.5,2x,f10.5)
       stop
       end
```

5/ ALGORITHMES UTILISES

5.1/ GENERALITES

La puissance de calcul de BLAISE repose sur une vaste bibliothèque de programmes numériques directement accessible par des commandes simples. Une grande partie des programmes de cette bibliothèque est directement tirée des bibliothèques suivantes: EISPACK, LINPACK, etc. (Voir les REFERENCES, page 68.)

On décrira sommairement dans ce paragraphe les principaux algorithmes implantés dans BLAISE. Ce chapitre sera plus détaillé dans les versions ultérieures.

5.1.1/ Algèbre Linéaire:

Les algorithmes classiques d'algèbre linéaire proviennent de la bibliothèque Linpack. On trouve les algorithmes de factorisation LU, QR, SVD et RREF pour la résolution de systèmes linéaires ou la détermination de bases de l'image et du noyau d'une matrice. Le rang d'une matrice est déterminé par SVD. Les programmes appelés dépendent naturellement du type réel ou complexe de la matrice.

Diverses commandes (calcul des inverses, des déterminants, de conditionnement, des bases orthonormées, de pseudo-inverses, ...) découlent directement de ces algorithmes.

Sont disponibles également la mise sous forme de Hessenberg et la factorisation de Cholesky pour les matrices symétriques ou hermitiennes (voir page 63).

La résolution des équations de Liapounov et Sylvester se fait par une méthode de Hessenberg-Schur.

5.1.2/ Analyse Spectrale:

Cette partie provient d'une partie des algorithmes de EISPACK et repose essentiellement sur la mise en forme de Schur réelle ou complexe. Les cas symétriques ou hermitiens sont traités de façon spécifique. D'autre part, d'un algorithme de détermination de la forme de Schur ordonnée (qui est utile pour la résolution des équations de Riccati) et d'un algorithme de bloc-diagonalisation (Bavely et Stewart).

5.1.3/ Fonctions matricielles.

Quelques fonctions matricielles (sin, cos, log, ...) sont disponibles et sont (volontairement) limitées au cas symétriques ou hermitien, et sont évaluées à partir de la forme diagonale de la matrice.

La fonction exponentielle est valide dans tous les cas (bloc-diagonalisation et approximants de Padé sur les blocs) (voir page 66).

5.1.4/ Faisceaux de matrices.

L'algorithme QZ est implanté dans BLAISE et permet le calcul de valeurs propres d'un faisceau s*E-A, ainsi que la forme de Schur généralisée (voir page 61).

5.1.5/ Optimisation.

Des algorithmes de minimisation sans contraintes et avec contraintes de bornes sont disponibles (commande optim). La fonctionnelle à minimiser peut être donnée en langage BLAISE ou par un sous-programme FORTRAN. (Voir Appendice B.)

5.1.6/ Simulation non linéaire.

Plusieurs programmes pour la résolution d'une équation differentielle (explicite ou implicite) sont implantés dans BLAISE. Pour les équations différentielles ordinaires, l'utilisateur peut soit spécifier la méthode d'intégration (Adams ou Gear), soit faire appel à un programme qui choisira la méthode à employer de façon automatique. (Voir Appendice C.)

5.2/ ANALYSE SPECTRALE

5.2.1/ Forme de Schur réelle (FSR):

Une matrice A est dans la forme de Hessenberg supérieure si, pour i > j + 1, on a A(i, j) = 0. La matrice carrée est bloc-triangulaire supérieure si elle peut être partitionnée de la façon suivante:

$$A = \begin{pmatrix} A(1,1) & A(1,2) & \cdots & A(1,n) \\ & & & \\ 0 & A(2,2) & \cdots & A(2,n) \\ & & & \\ \cdots & \cdots & \cdots & \cdots \\ & & & \\ 0 & 0 & \cdots & A(n,n) \end{pmatrix} \qquad (1)$$

où chaque bloc situé sur la diagonale est carré. Si l'ordre de chaque bloc situé sur la diagonale n'excède pas 2, alors on dit que A

est sous forme quasi-trangulaire. Dans le cas où A est quasi-triangulaire supérieure et les blocs de taille 2x2 qui sont sur la diagonale correspondent seulement à de paires complexes de valeurs propres, on dit que A est dans la forme de Schur réelle (FSR).

La réduction d'une matrice à une forme quasi-triangulaire ou à une FSR est la technique de base pour le calcul numérique des valeurs propres et des vecteurs propres d'une matrice réelle générale.

La méthode la plus employée pour calculer la FSR d'une matrice est l'algorithme double QR de Francis. Cet algorithme travaille directement avec une matrice réelle en forme de Hessenberg supérieure. La réduction combinée à la forme de Hessenberg et à la FSR est numériquement stable puisque resultant de l'accumulation de transformations orthogonales. Dans BLAISE, les subroutines d'EISPACK, ORTHES et ORTRAN sont employées pour calculer la forme de Hessenberg et accumuler les transformations. HQR0R2 est une version plus courte de la subroutine HQR2, issue aussi d'EISPACK, qui calcule la FSR d'une matrice.

Pour le calcul des valeurs propres on a ajouté l'option d'équilibrage offerte par la bibliothèque EISPACK. La subroutine BALANC minimise la norme l1 par changements de base avec des matrices diagonales non-singulières. On choisit les éléments de la diagonale comme étant des puissances de la base numérique de la machine. BALANC isole aussi les valeurs propres individuelles par des permutations de lignes et colonnes.

5.2.2/ Ordonnancement de la FSR:

Une proprieté importante de la FSR est que deux blocs diagonaux consécutifs peuvent être permutés en employant un changement de base orthogonal. Ainsi, on peut grouper de façon arbitraire les valeurs propres afin de déterminer des bases pour les sous-espaces invariants d'une matrice. Par exemple, les vecteurs propres orthogonaux qui correspondent aux valeurs propres situés dans les i premières positions de la diagonale d'une matrice donnée dans la forme de Schur réelle, fournissent une base orthogonale pour le sous-espace invariant de la matrice correspondante associé à cet ensemble de i valeurs propres. Etant donnée une collection quelconque de i valeurs propres il est possible de transformer la FSR de telle façon que la matrice obtenue ait ces valeurs propres dans les position initiales au moyen de changements de base orthogonaux. Les i premières colonnes de la matrice de transformée, nous donnent une base orthogonale du sous-espace invariant qui corresponde aux i valeurs propres choisies.

Deux stratégies d'ordonnancement sont implantées dans BLAISE, de façon à avoir une base orthogonale du sous-espace stable d'une matrice réelle dans les cas "continu" (parties réelles negatives) ou "discret" (intérieur du cercle unité).

5.2.3/ Extension au cas d'un faisceau:

Les techniques décrites plus haut pour le calcul de la FSR d'une matrice et pour la mise en ordre de ses blocs diagonaux ont été étendus à des faisceaux de matrices.

Soient A et B matrices réelles quelconques du même ordre, qui définissent le problème spectral généralisé:

$$Ax = (\lambda)\ Bx$$

L'algorithme QZ, qui a été décrit en détail par Moler et Stewart [9b], permet la réduction simultannée de A à une forme de Schur réelle supérieur et de B à une forme triangulaire supérieure. Nous appelerons cela: réduction à la forme FSR-triangulaire.

Le principe essentiel de l'algorithme QZ est d'appliquer le pas QR à la matrice AB**(-1) sans la calculer explicitement. L'algorithme QZ consiste en quatre étapes, mises en oeuvre par des subroutines séparées qui devront être appelées les unes après les autres. Ces subroutines sont: QHESZ, QITZ, QVALZ et QVECZ. La subroutine QHESZ réduit simultanément A à la forme de Hessenberg supérieure et B à la forme triangulaire supérieure en employant des transformations orthogonales de lignes et de colonnes.

A partir d'une paire de matrices de Hessenberg triangulaires, QITZ réduit A à une forme quasi-triangulaire, tout en conservant la structure triangulaire supérieure de B. Alors, QVALZ réduira la matrice A à une FSR supérieure, en conservant à B sa forme triangulaire, et calculera les quantités qui donneront les valeurs propres généralisées. En utilisant la sortie de QVALZ, la subroutine QVECZ calcule les valeurs propres généralisées. Ces subroutines ont été obtenues par des modifications des subroutines EISPACK correspondantes; toutes accumulent les transformations orthogonales par colonnes produite par l'exécution du processus de réduction. Les subroutines QHESZ, QITZ et QVALZ peuvent aussi accumuler les transformations de lignes qui ont été appliquées. Cette facilité est importante pour certaines applications et elle n'est pas incorporée dans la bibliothèque EISPACK.

Il est possible de permuter deux paires consécutives de blocs diagonaux d'une paire de matrices FSR-triangulaires en employant des transformations orthogonales des lignes et des colonnes. Ainsi, on peut obtenir des groupements arbitraires de valeurs propres généralisées et produire des bases pour des sous-espaces associés à certaines valeurs propres généralisées. Une application de cette technique se trouve dans la résolution des équations de Riccati presque-singulières.

La subroutine EXCHQZ échange deux paires consecutives de blocs diagonaux d'une paire FSR-triangulaire. Pour des blocs diagonaux 2x2 on applique l'algorithme QZ, avec une mise en oeuvre particulière (voir Van Dooren [12]). On emploie, seulement de rotations planes de lignes et colonnes. A travers l'emploi de cette subroutine, deux strategies différentes et utiles pour la séparation

du spectre dans le problème spectral généralisé sont inclues dans BLAISE.

5.2.4/ Calcul de la forme bloc-diagonale:

La réduction d'une matrice à la forme bloc-diagonale:

$$X^{-1} A X = D = \mathrm{diag}(D1, D2, \ldots, DS) \qquad (2)$$

a des multiples applications. Quand les blocs sont petits, on peut calculer les puissances de A économiquement, de la forme suivante:

$$A^K = X\, \mathrm{diag}(D1^K, D2^K, \ldots, DS^K)\, X^{-1}$$

et ce fait peut être employé pour simplifier le calcul des fonctions de A définies par des séries des puissances, par exemple, l'exponentielle d'une matrice. Si on fait la partition d'une matrice X d'après sa structure bloc-diagonale, dans la forme:

$$X = (X1, X2, \ldots, XS)$$

alors les colonnes de X1, par exemple, forment une base pour un sous-espace réduisant de A et D1 est la représentation de A par rapport à cette base. La forme bloc-diagonale peut être employée pour réduire la matrice à la forme normale de Jordan.

La réduction d'une matrice FSR à la forme bloc-diagonale est exécutée par la subroutine BDIAG. La clé du problème consiste à mantenir la matrice de transformation si bien-conditionnée que possible. L'algorithme de bloc-diagonalisation est décrit en détail dans Bavely et Stewart [?] et les grandes lignes sont décrites dans la suite.

Soit A une matrice FSR, partitionnée de la façon suivante:

$$A = \begin{vmatrix} A11 & A12 \\ 0 & A22 \end{vmatrix}$$

où la taille de A11 est 1x1 ou 2x2 selon la dimension du premier bloc diagonal de A. On essaye, alors de trouver un changement de base tel que:

$$X^{-1} AX = \begin{vmatrix} A11 & 0 \\ 0 & A22 \end{vmatrix}$$

Si on peut trover une telle transformation et que la matrice transformée n'est pas trop mal-conditionnée, la réduction continue a s'effectuer sur la matrice A22. Dans le cas contraire, on trouve un bloc convenable sur A22 qui soit de taille 1x1 ou 2x2 et on le déplace par des transformations orthogonales à la position principale, c'est à dire à la première position sur la diagonale

principale de A22. Le bloc est alors joint à A11 en augmentant l'ordre de A11 en une valeur appropiée, a savoir 1 ou 2, et on fait un autre essai pour trouver une matrice de transformation X.

La transformation X est calculée de la façon suivante:

$$X = \begin{vmatrix} I & P \\ 0 & I \end{vmatrix}$$

où les ordres des matrices identité sont les mêmes que ceux de A11 et A22. P est la solution de l'équation de Sylvester suivante:

$$A11 * P - P * A22 = -A12$$

La solution de cette équation est obtenue avec la subroutine SHRSLV. Comme la matrice X est mal-conditionnée quand P est grande, la subroutine SHRSLV examine chaque élément de P qu'elle génère pour voir si sa grandeur excede une borne supérieure fournie par l'utilisateur. Si c'est le cas, on abandone l'essai de calcul de X et on forme un bloc A11 plus grand. Par contre, si aucun élément de P n'excède la borne, on accepte la matrice X et A est réduite de la façon déja décrite. La transformation X est post-multipliée dans une matrice où l'on accumule toutes les transformations faites sur la matrice A.

5.2.5/ Equations matricielles de Sylvester et Lyapounov:

Considérons les équations de Sylvester suivantes:

$$AX + XB = C \qquad (1)$$

$$AXB + X = C \qquad (2)$$

où A, B et C sont matrices réelles de tailles mxm, nxn et mxn, respectivement, et X est une matrice inconnue de taille mxn. On trouve assez frequemment des telles équations dans les applications. Par exemple, la réduction d'une matrice à la forme bloc diagonale exige la solution répétée d'une équation (1) (Voir page 63).

Des cas importants des équations (1) et (2) sont les équations matricielles de Lyapounov suivantes:

$$A'X + XA = C \qquad (3)$$

$$A'XA + X = C \qquad (4)$$

que l'on trouve dans les critères de stabilité et dans la solution itérative des équations matricielles de Riccati (voir page 65). L'équation (3) est connue sous le nom d'équation "continue" de Lyapounov, puisqu'on la retrouve dans le traitement de systèmes dynamiques continus dans le temps. Pour une raison similaire l'équation (4) est appelée équation de Lyapounov "discrète". Les termes "continu" et "discret" sont employés aussi pour les équations de Sylvester (1) et (2), respectivement.

Une méthode numérique efficace et sûre pour résoudre ces équations est la méthode de transformation: l'équation originale est réduite en se servant de transformations orthogonales à une équation similaire, laquelle peut être résolue facilement par une procedure de substitution. Alors, la solution de l'équation réduite est transformée inversement pour obtenir la solution de l'équation originale.

Pour la résolution des équations de Lyapounov, on réduit en premier la matrice A à la forme quasi-triangulaire supérieure. Soit U une matrice de transformation orthogonale qui réalise une telle réduction. Prémultipliant par la tranposée de U et post-multipliant par U l'équation (3) (ou la (4)), on obtient une équation semblable où la matrice A est remplacée par sa forme quasi-triangulaire correspondante. En prennant en compte le partitionnement implicite dans la structure quasi-triangulaire, la solution de cette équation de Lyapounov réduite est obtenue par la résolution successive, dans un ordre précis, de systèmes linéaires dont l'ordre n'excede pas 4. Cette technique peut être étendue aux équations de Sylvester. Dans ce dernier cas, les deux matrices A et B sont réduites à la forme quasi-triangulaire supérieure. On peut trouver des détails sur les algorithmes dans Bartels et Stewart [1].

Une autre méthode de transformation pour les équations de Sylvester, connue sous le nom de méthode de Hessenberg-Schur, a été proposé par Golub, Nash et Van Loan [4]. Dans cette méthode, on réduit la matrice A à la forme supérieure de Hessenberg supérieure et la transposée de la matrice B à la forme trangulaire supérieure. La solution des équations de Sylvester transformées s'obtient en résolvant successivement des systèmes linéaires d'ordre m ou 2m, dont les matrices de coefficients sont triangulaires supérieures avec une ou deux sous-diagonales non-nulles, respectivement. On effectue la résolution de ces systèmes linéaires en employant l'élimination de Gauss avec pivotement partiel. La méthode de Hessenberg-Schur est plus efficace que la méthode de "Schur", dont on a parlé plus haut, mais elle demande environ 2m*m places additionelles de mémoire pour les données (les matrices de coefficients peuvent être gardées de façon compressée). Dans certains cas, comme par exemple pour la bloc-diagonalisation d'une matrice, A et B sont déja FSR et il est plus avantageux appliquer l'algorithme de substitution rétrograde décrit en [1].

Les subroutines LYBSC et LY3AD résolvent les équations matricielles de Lyapounov (3) et (4), respectivement, dans lequelles on suppose que A est en forme quasi-triangulaire supérieure. Pour des équations de Lyapounov en général, A doit être préalablement réduite à la forme quasi-triangulaire supérieure, au moyen de ORTHES, ORTRAN, et HQROR2.

La subroutine SYHSC résout l'équations matricielle de Sylvester (1), où A est sous forme quasi-triangulaire inférieure et B est en forme quasi-triangulaire supérieure en employant la méthode de Heisenberg-Schur.

5.2.6/ Equations de Riccati:

Dans l'analyse et la conception de systèmes de contrôle optimal les problèmes d'estimation, ainsi que dans autres branches de la mathématiques appliquées, il est nécessaire de résoudre les équations matricielles algébriques quadratiques de Riccati suivantes:

$$A' X + X A - X B R^{-1} B' X + Q = 0 \qquad (1)$$

$$X = A' X A - A' X B (R + B' X B)^{-1} B' X A + Q \qquad (2)$$

où A, B, Q et R sont des matrices réelles données de tailles nxn, nxm, nxn et mxm, respectivement, et X est une matrice réelle inconnue de taille nxn. On suppose que Q est semi-définie positive et R est définie positive et que les conditions nécessaires et suffisantes pour l'existence et l'unicité d'une solution X semi-définie positive sont remplies. Ces conditions sont, par exemple, que les paires (A, B) et (A',sqrt(Q)) soient stabilisables. L'équation (1) est connue comme l'équation de Riccati continue puisqu'elle apparaît dans le traitement des systèmes dynamiques continus. Pour une raison similaire, l'équation (2) est appelée l'équation discrète. La solution semi-définie positive des équations de Riccati algébriques peut être employée pour l'obtention de la stabilisation optimale des systèmes dynamiques linéaires.

La méthode numérique la plus efficace et sûre pour résoudre les équations de Riccati algébriques est probablement la méthode directe proposée par Laub [5]; la solution de ces équations est obtenue par l'emploi des vecteurs de Schur associés aux valeurs propres stables d'une certaine matrice 2nx2n bâtie en fonction des matrices données A, B, Q et R. Pour le cas continu, la matrice étendue est donnée par:

$$G = \begin{vmatrix} A & -B R^{-1} B' \\ -Q & -A' \end{vmatrix}$$

(où, comme d'habitude A' est la transposée de A). Cette matrice est premièrement réduite à FSR et, alors les valeurs propres stables et les vecteurs de Schur correspondants sont séparés des instables. Les valeurs propres stables sont groupées dans le coin supérieur gauche de la matrice FSR résultante. Soit U la matrice de transformation qui réalise la réduction de la matrice originale à la matrice finale qui a ses blocs diagonaux ordonnés. Alors la solution obtenue de l'équation de Riccati est:

$$U11' X = U21'$$

où Uij est le bloc i,j-ième de taille nxn, qui est dans les n premières colonnes de U. Dans le cas discret la procédure est similaire.

Pour résoudre les équations de Riccati algébriques discrètes avec une matrice singulière A, ou pour résoudre les équations de Riccati continues ou discrètes avec une matrice quasi-singulière R on doit employer une autre méthode directe, décrite par Van Dooren [12]. Dans cette méthode on emploie les vecteurs de Schur généralisés à la place des vecteurs de Schur ordinaires. On obtient deux matrices G et F de tailles (2n+m)x(2n+m) en fonction des matrices données A, B, Q et R. Contrairement à la méthode des vecteurs de Schur décrite plus haut, pour former F et G il n'est pas necessaire d'effectuer des inversions de matrices. G et F définissent un problème spectral généralisé réel d'ordre 2n+m. Ce problème spectral est alors réduit à un problème spectral généralisé d'ordre 2n équivalent par l'application de transformations orthogonales de ligne. Les vecteurs de Schur généralisés qui correspondent aux valeurs propres généralisés stables sont employés dans le calcul de la solution de l'équation de Riccati, comme on l'a vu plus haut.

5.2.7/ Simulation de systèmes linéaires:

La méthode pour le calcul l'exponentielle d'une matrice est basée sur la décomposition bloc-diagonale de A par un changement de base de la forme:

$$A = Q\ D\ Q^{-1}$$

où D est une matrice bloc-diagonale. Chaque bloc en D comprend un groupement de valeurs propres voisines. La définition de exp(tA) par de séries de puissances nous donne:

$$\exp(tA) = Q\ \exp(tD)\ Q^{-1} \qquad (1)$$

et exp(tD) a la même structure de blocs que D. L'exponentielle des blocs diagonaux est calculée par la subroutine PADE, laquelle prend en compte que la forme bloc-diagonale obtenue de BDIAG est aussi une FSR.

La clé du problème dans le calcul de l'exponentielle d'une matrice par des méthodes de décomposition est le besoin de contrôler le conditionnement de la matrice de transformation cond(Q) (voir page 26). Un choix de Q tel que cond(Q)<100 implique, en gros, que pas plus de deux chiffres décimaux significatifs seront perdus comme consequence des erreurs d'arrondi chaque fois que exp(tA) est obtenu à partir de exp(D) en se servant de (1). Une borne plus grande entraine la perte de plus de chiffres et un gain de temps de calcul du à la taille plus petite des blocs de D. Dans la pratique, nous devrons attendre que tous les blocs soient 1x1 ou 2x2 et, ainsi, les temps de calcul de exp(tD) seront très petits. La technique de décomposition est mise en oeuvre par les subroutines DEXPM1 ou WEXPM1 dans le cas complexe.

Ces subroutines peuvent être employées dans la determination du système échantillonné correspondant à un système continu de la forme:

$$dx(t)/dt = Ax(t) + Bu(t) \qquad (2)$$

où u(t) est une fonction de contrôle donnée. Soit s la période d'échantillonnage et supposons que u a une valeur constante u(k) sur chaque intervale [ks, (k+1)s]. Nous appelons x(k) la valeur de x au point t=ks. Le système échantillonné correspondant à (2) est

$$x(k + 1) = F(s)x(k) + H(s)u(k)$$

où

$$F(s) = \exp(A * s), \text{ et } H(s) = \int_0^s \exp(A * t) * B * u(t)\, dt$$

Si nous définissons la matrice

$$G = \begin{vmatrix} A & B \\ 0 & 0 \end{vmatrix}$$

les matrices F(s) et H(s) résultent de l'identité

$$\exp(G * s) = \begin{vmatrix} F(s) & H(s) \\ 0 & I \end{vmatrix}$$

La méthode de discretisation précédente est due à Van Loan [14].

6/ REFERENCES.

[1].- BARTELS, R.H. and STEWART, G.W., Solution of the equation AX+XB=C, Comm. ACM, vol. 15, p 820-826, 1972.

[2].- BAVELY, C.A. and STEWART, G.W., An algorithm for computing reducing subspaces by block diagonalisation, SIAM J. Numer. Anal., vol. 10, p. 359-367, 1979.

[3].- EISPACK Guide, Lecture Notes in Comp. Science, vol. 6, Springer, 1976.

[4].- GOLUB, G.H., NASH, S. and VAN LOAN, C.F., A Hessenberg-Schur method for the problem AX+XB=C, IEEE Trans. Autom. Control, vol AC-24, p. 909-913, 1979.

[5].- LAUB, A.J., A Schur method for solving the algebraic matrix Riccati equations, IEEE Trans. Autom. Control, vol AC-24, p. 913-921, 1979.

[6].- LEMARECHAL, C., La norme Modulopt, INRIA, 1980.

[7].- LINPACK User's Guide, SIAM, Philadelphia, 1979.

[8].- MARROCCO, A., Modulef: Module Tracou, INRIA, 1977.

[9a].- MOLER, C., MATLAB User's Guide, Tech. Rep. CS81-1, Dept. of Computer Sci., Univ. New Mexico, August, 1982.

[9b].- MOLER, C.B. and STEWART, G.W., An algorithm for generalized eigenvalue problems, SIAM J. Num. An., vol. 10, p. 241-256, 1978.

[10].- PETZOLD, L., Automatic selection of methods for solving stiff and non-stiff systems of ode, Sandia Labs, Report 8230.

[11].- ROCHE, J.-R., Application des approximants de Padé au calcul de l'exponnentielle d'une matrice, Th. 3e. cycle, USMG, Grenoble, 1980.

[12].- VAN DOOREN, P., A generalized eigenvalue approach for solving the Riccati equations, Report NA-80-02, Comp. Sci. Dept., Stanford Univ., 1980.

[13].- VAN DOOREN, P., The generalized Eigenstructure Problem in Linear System Theory, IEEE-AC, vol. 26, 1981.

[14].- VAN LOAN, C.F., Computing integrals involving matrix exponentials, IEEE Trans. Autom. Control, vol AC-23, p. 395-404, 1978.

[15].- WARD, C.B., Numerical computation of the matrix exponential with accuracy estimate, SIAM J. Numer. An., vol. 14, p. 600-610, 1977.

[16].- WIRTH, N., Algorithms + Data Structures = Programs, Prentice-Hall, Englewood Cliffs, N.J., 1976.

APPENDICE A

VECTEUR DESS

TABLE DES MATIERES

TABLEAU DES CORRESPONDANCES

ifi	1
istand	2
rdon(1:16)	3:18
idon(1:8)	19:26
ibouc1	27
ilist	28
iplac	29
iec	30
ica	31
vminx	32
vmaxx	33
vminy	34
vmaxy	35
kplum	36
igrad	37
iaxsuj	38
modgrx	39
xint	40
ngradx	41
modgry	42
yint	43
ngrady	44
inumx	45
modulx	46
inumy	47
moduly	48
legcou	49
xlong	50
ilmult	51
ylong	52
nucou	53
isupe	54
ichoix	55
ichar	56
dime	57
pas(1:10)	58:67
np	68
homx	69
homy	70
nbloc	71
nunit	72

1. PARAMETRES GENERAUX DE DEFINITION DU DESSIN

ISTAND, RDON(16), IDON(8), IBOUCL, ILIST, IPLAC, EIC, ICA

A - ISTAND = DESS(2)

Option de cadrage et de mise en page du graphique

Les paramètres directement en relation sont les éléments des tableaux RDON et IDON dont nous verrons ci-dessous la signification précise.

Quatre valeurs sont possibles pour le paramètre ISTAND

ISTAND = 0
ISTAND = 1 ← valeur donnée par défaut
ISTAND = 2
ISTAND = -1

ISTAND = 1

Le programme affecte les valeurs des tableaux RDON et IDON (1.1, 1.2). Les dimensions initiales du dessin sont standard

33,6 cm × 47,5 cm

Le cadre sujet, contenant la ou les courbes à visualiser est placé par le programme et a pour dimensions

25 cm × 35 cm

Les valeurs minimum et maximum déterminant la graduation du cadre sujet sont respectivement les valeurs minimum et maximum de

VX(I) et VY(I) pour I ∈ [NP1, NP2].

ISTAND = 2

Analogue à ISTAND = 1 sauf que l'utilisateur précise lui-même les bornes du cadre sujet par l'intermédiaire des variables

VMINX, VMAXX, VMINY, VAMXY

ISTAND = 0

Dans ce cas, la mise en page, les dimensions initiales du dessin, la disposition du cadre sujet, le choix de la graduation (linéaire ou logarithmique, ...) est laissée aux soins de l'utilisateur, qui doit alors préciser les valeurs des éléments des tableaux RDON et IDON. Notons aussi, que ces tableaux sont initialisés aux valeurs indiquées dans (1.1, 1.2).

ISTAND = -1

Analogue à ISTAND = 0, sauf que les valeurs de cadrage VMINX, VMINY, VMAXX, VAMXY sont calculées automatiquement. C'est l'option par défaut.

1.1. Signification des éléments de RDON(1:16) = DESS(3:18)

RDON(i) = DESS(2+i)

- Définition des cadres - valeurs en cm

	Valeurs imposées ISTAND = 1 ou 2	INITIALISATION par défaut
RDON(1) abscisse (objet) début cadre sujet	6.	6.
RDON(2) abscisse (objet) fin cadre sujet	31.	41.
RDON(3) ordonnée (objet) début cadre sujet	9.	9.
RDON(4) ordonnée (objet) fin cadre sujet	44.	44.
RDON(14) marge entre cadres sujet/objet-haut	3.5	3.5
RDON(15) marge entre cadres sujet/objet-droite	2.6	2.6
RDON(16) espace entre dessins successifs	.3	.3

- Valeurs déterminant le cadre sujet

	Valeurs imposées ISTAND = ±1 ou 2 (*)	INITIALI-SATION par défaut
RDON(5) abscisse (sujet) bord gauche	min Vx(I)	-1.
RDON(6) ordonnée (sujet) bord inférieur	min Vy(I)	-1.
RDON(7) abscisse (sujet) bord droit	max Vx(I)	+1.
RDON(8) ordonnée (sujet) bord supérieur	max Vy(I)	+1.

(*) Correspond à ISTAND = 11 ; dans le cas de graduation logarithmique, on prend la puissance de 10 immédiatement inférieure ou supérieure selon le cas.

- Autres paramètres réels en cm

	Valeurs imposées ISTAND = 1 ou 2 (*)	INITIALI-SATION par défaut
RDON(9) largeur des caractères	0.4	0.4
RDON(10) hauteur des caractères	0.5	0.5
RDON(11) longueur tirets sur bord x	0.5	0.5
RDON(12) longueur tirets sur bord y	0.4	0.4
RDON(13) espace entre bord y et début graduation	2.6 ou 1.4 ou 3.2	3.2

1.2. Signification des éléments de IDON (1:8) = DESS (19:26)

IDON(i) = DESS(18+i)

	Valeurs imposées ISTAND = 1 ou 2	INITIALI-SATION par défaut
IDON(1) échelle sur axe x	(**)	0
IDON(2) échelle sur axe y		0
IDON(3) Nombre d'intervalles référencés sur bord x du cadre sujet (cas échelle linéaire). Dans le cas d'échelle log, le nombre de modules est calculé par le programme.	5(***)	2
IDON(4) Idem bord y	10	2

	Valeurs imposées ISTAND = 1 ou 2	INITIALISATION par défaut
IDON(5) Nombre de sous-intervalles en x (Cas échelle linéaire)	10	10
IDON(6) Nombre de sous-intervalles en y (Cas échelle linéaire)	5	10

L'exemple de présentation générale donnée en 1.3 correspond à :

IDON(1) = 0
IDON(2) = 0
IDON(3) = 5
IDON(4) = 4
IDON(5) = 11
IDON(6) = 2

(**) IDON(1) = 1 (IDON(2) = 1) correspond à échelle logarithmique sur l'axe correspondant. L'échelle est linéaire pour valeur ≠ 1

L'échelle logarithmique est prise dans le cas ISTAND = 1 ou 2 si :

a) RDON(5) (ou RDON(6)) > 0

b) $\text{Log}_{10} \left(\frac{v_{max}}{v_{min}}\right) \geq 3$

(***) Dans le cas MODGRX = 1 ou 2, (MOGDRY = 1 ou 2)
IDON(3), IDON(4), IDON(5), IDON(6) n'ont plus de signification et ne sont pas utilisés par le programme.

IDON(7) x
sont les formats BENSON d'écriture des nombres
IDON(8) y

Le choix retenu dans le cas ISTAND = 1 ou 2 et dans le cas d'échelle linéaire, est le suivant :

On pose $v = \max (|v_{max}|, |v_{min}|)$

a) $v \leq 1$ → valeur 3

b) $1 < v \leq 10$ → valeur 2

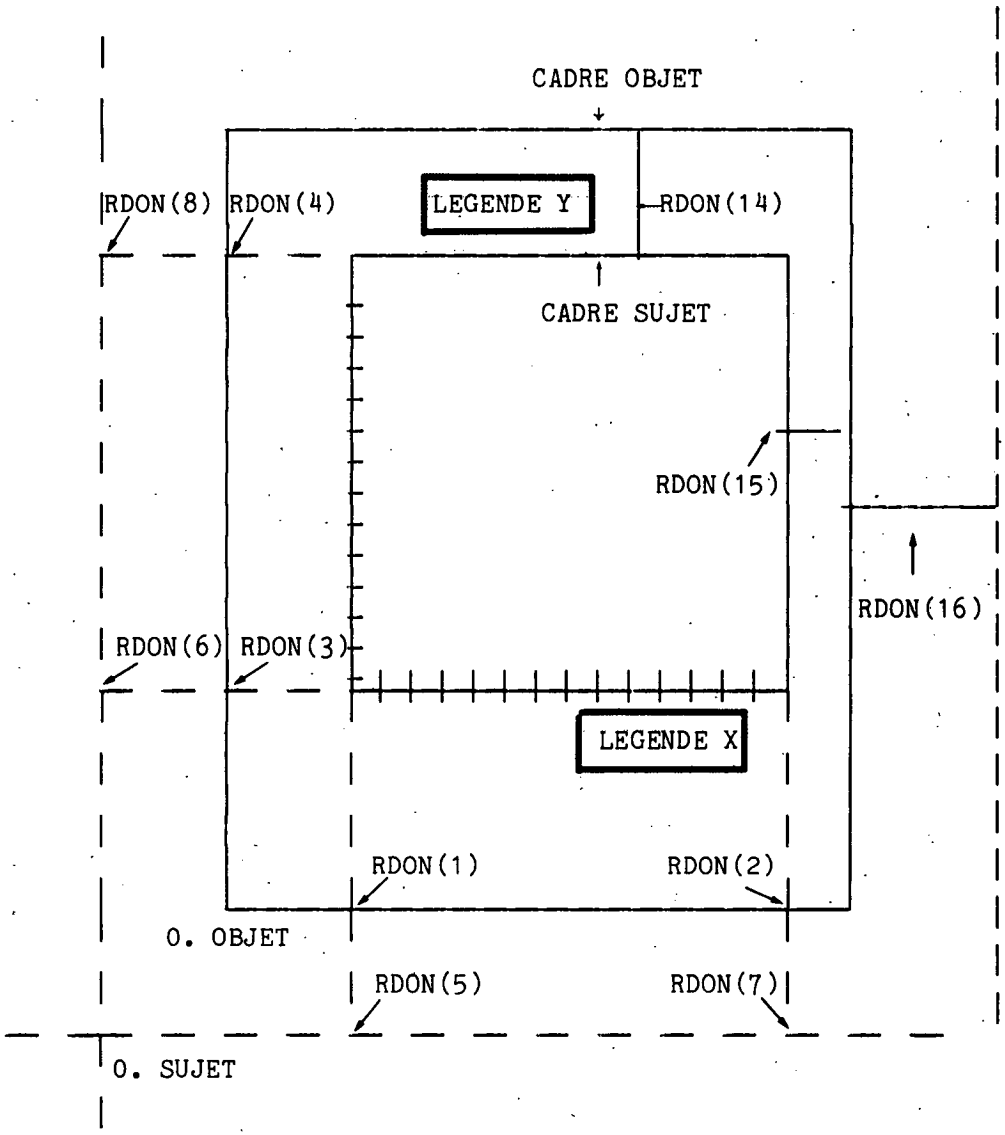
c) $10 < v \leq 100$ → valeur 1

d) $v \leq 1000$ → valeur -1

e) pour axe y seul → valeur -2 si $v \geq 1000$
ou si $v < 0.01$.

Valeurs par défaut : IDON(7) = -1
IDON(8) = -1

1.3 Présentation générale d'un dessin



Présentation générale d'un dessin. Deux repères se superposent, le repère SUJET (coordonnées utilisateur) et le repère OBJET (coordonnées en cm)

B. IBOUCL = DESS(27)

Option permettant lors de la superposition de courbes ISUPE = 1, d'éviter le retour de la plume à l'origine objet.

Deux valeurs sont possibles pour le paramètre IBOUCL

IBOUCL = 0 ← Valeur donnée par défaut
IBOUCL = 1

IBOUCL = 0

C'est l'option par défaut ; toutes les courbes, ainsi qu'éventuellement les légendes correspondantes se trouvent dans des blocs différents.

IBOUCL = 1

On évite le retour à l'origine objet et toutes les courbes, ainsi que les légendes se trouvent dans un même bloc. Cette option est utile lorsque le dessin est constitué de nombreuses courbes, ou lorsque la courbe est constuite par morceaux.

C. ILIST = DESS(28)

Option permettant de retrouver sur le listing la valeur des options et des paramètres.

Ce paramètre prend essentiellement 3 valeurs

ILIST = 0 (≤ 0)
ILIST = 1 ← valeur donnée par défaut
ILIST = 2 (≥ 2).

ILIST = 1

On imprime sur le listing la valeur des différents paramètres et options retenus ainsi que les numéros de blocs.

ILIST = 2

On imprime en plus les valeurs des tableaux VX et VY constituant la courbe à tracer.

ILIST = 0

On n'imprime rien sur le listing.

D. IPLAC = DESS(29)

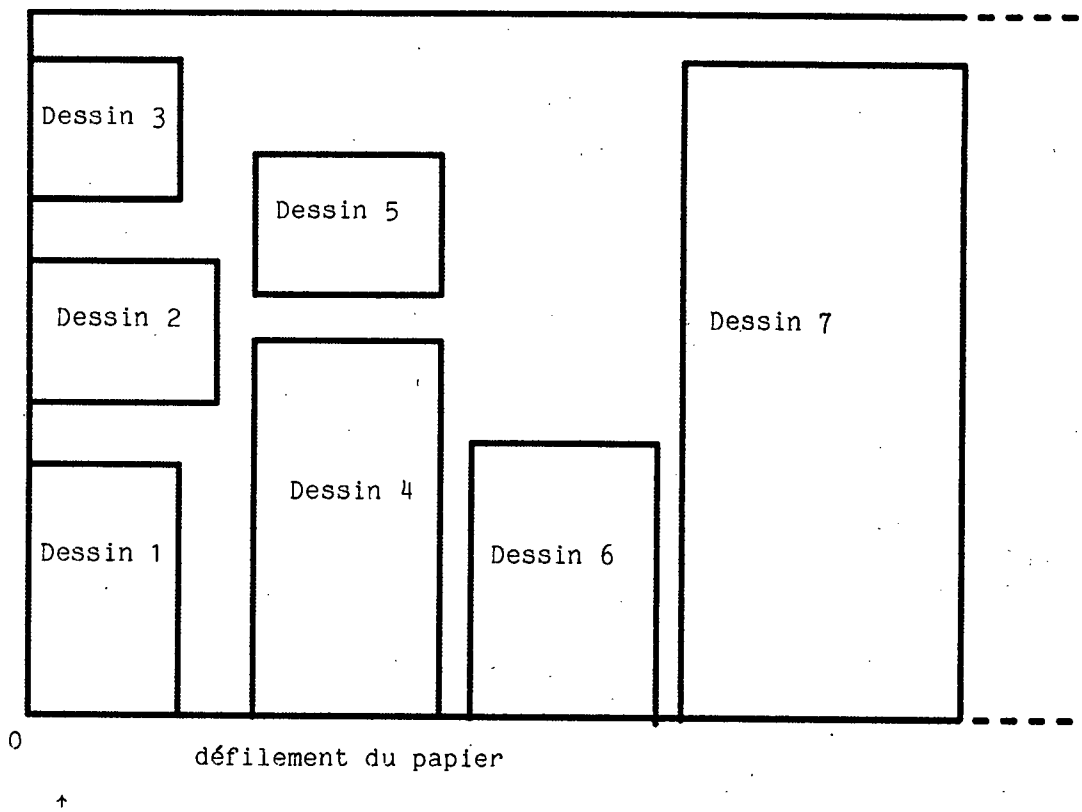
Option de cadrage automatique des dessins sur dérouleur BENSON. Deux valeurs possibles pour le paramètre IPLAC

IPLAC = 0
IPLAC = 1 ← valeur donnée par défaut.

IPLAC = 1

Les dessins se placent automatiquement sur une feuille de papier de largeur l (l initialise à 70 cm par défaut).

Il faut placer correctement la plume épart.



IPLAC = 0

Il n'y a pas de changement d'origine objet et tous les dessins sont générés à la même place.

E. IEC = DESS(30)

Epaisseur de tracé des cadres sujet et objet.

Initialisé à 1 par défaut

$1 \leq IEC \leq 7$

F. ICA = DESS(31)

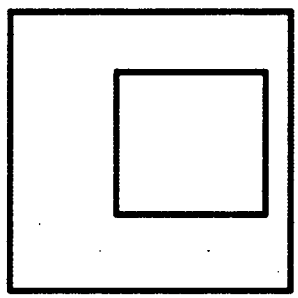
Option relative au tracé du cadre sujet.

Trois valeurs essentielles pour cette option ICA

ICA = -1	(< 0)	
ICA = 0		valeur donnée par défaut
ICA = 1	(> 0)	

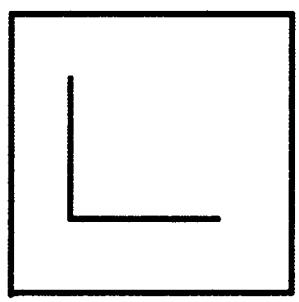
ICA = 0

On trace le cadre sujet entièrement.



ICA = 1

On ne trace qu'une partie du cadre sujet, analogue à des axes.



ICA = -1

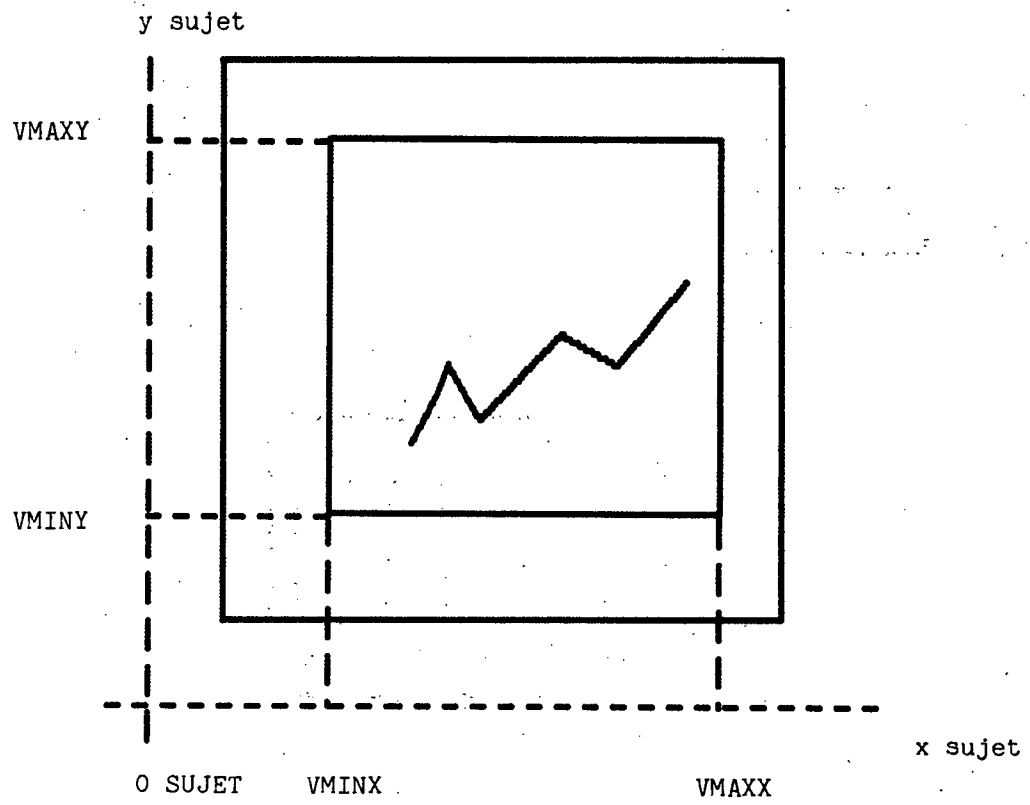
On ne trace pas le cadre, et seule la graduation est apparente.

2. PARAMETRES DE DEFINITION DU CADRE SUJET LORSQUE ISTAND = 2

VMINX,VMAXX,VMINY,VMAXY

Lorsque ISTAND = 2 on peut chosir son cadre sujet par l'intermédiaire des variables

VMINX = DESS(32)
VMINY = DESS(33)
VMAXX = DESS(34)
VMAXY = DESS(35)



3. PARAMETRES RELATIFS AU TRACE

KPLUM = DESS(36)

Paramètre désignant le n° de la plume utilisée (dans le cas de BENSON à plusieurs plumes). KPLUM initialisé à 0 par défaut.

4. PARAMETRES LIES A LA GRADUATION DU CADRE SUJET

```
IGRAD,IAXSUJ,
MODGRX, XINT, NGRADX,
MODGRY, YINT, NGRADY,
INUMX, MODULX,
INUMY, MODULY
```

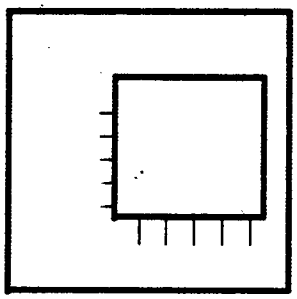
A. IGRAD = DESS(37)

Option relative au sens des tirets sur le bord du cadre sujet. Deux valeurs pour cette option IGRAD

IGRAD = 1 valeur donnée par défaut
IGRAD = 0

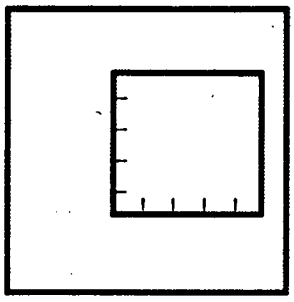
```
IGRAD = 1
```

La graduation sur le cadre sujet se fait ainsi



IGRAD = 0

Les tirets sont dans l'autre sens.



B. **IAXSUJ = DESS(38)**

Option permettant à l'utilisateur de tracer les axes sujets s'ils sont à l'intérieur du cadre sujet.

Trois valeurs pour cette option IAXSUJ

IAXSUJ = 0 ← valeur donnée par défaut
IAXSUJ = 1
IAXSUJ = 2

IAXSUJ = 0

On ne trace pas le ou les axes sujets, même s'ils sont visibles dans le cadre sujet.

IAXSUJ = 1

On trace le ou les axes sujets visibles dans le cadre.

IAXSUJ = 2

Il faut que l'origine sujet soit à l'intérieur du cadre sujet pour que le programme trace les axes.

C. MODGRX = DESS(41), MODGRY = DESS(42)

Option mode de construction de la graduation sur le bord x ou le bord y du cadre sujet.

Les paramètres directement en relation avec ces options sont XINT, NGRADX, YINT, NGRADY et IDON(I), I = 1,6.

Ces options ne sont effectives que dans le cas d'échelle linéaire sur les axes, c'est-à-dire :

IDON(1) = 0 pour MODRGX
IDON(2) = 1 pour MODGRY

Quatre valeurs pour cette option

MODGRX (ou MODGRY) = 0 par défaut
MODGRX (ou MODGRY) = 1
MODGRX (ou MODGRY) = 2
MODGRX (ou MODGRY) = 3

MODGRX = 0 (MODGRY = 0)

On partage le bord du cadre sujet x (ou y) en un nombre d'intervalles donnés.

Ce nombre est fixé à 5 (et 10) dans le cas des options ISTAND = 1 ou 2, mais ce nombre peut être choisi par l'utilisateur dans le cas ISTAND = 0 par l'intermédiaire du tableau IDON (voir 1.1 et 1.2).

MODGRX = 1 (MODGRY = 1)

On se donne un intervalle élémentaire, et la graduation se fait en portant cet intervalle de la valeur minimum à la valeur maximum du cadre.

Cet intervalle est identifié par XINT, (YINT) qui est donné en centimètres, et qui va représenter la longueur sur le dessin final de l'intervalle élémentaire en x (ou en y).

Tous les NGRADX, (NGRADY) on imprime la valeur de l'abscisse (ou ordonnée) correspondante.

Ce mode de construction n'est réalisable que dans le cas d'échelle linéaire. On peut s'imposer l'échelle linéaire avec ISTAND = 0, mais dans le cas ISTAND = 1 ou 2 le choix de l'échelle sur les axes est fait par le programme selon le procédé exposé en 1.2, et la possiblité d'utiliser MODGRX = 1, (MODGRY = 1) ne sera effective que si l'échelle correspondante retenue est linéaire.

MODGRX = 2 (MODGRY = 2)

Utilisation identique à MODGRX = 1, mais XINT, (YINT) toujours donnés en centimètres représentent la longueur des intervalle élémentaires sur le dessin initial. (Le dessin initial subit une homothétie en x et y avant de devenir le dessin final effectivement visualisé).

MODGRX = 3 (MODGRY = 3)

Même utilisation que précédemment, mais cette fois-ci XINT, (YINT) représentent les intervalles élémentaires exprimés en unités sujet.

D. INUMX = DESS(45), INUMY = DESS(47)

Option permettant de modifier l'écriture des valeurs sur les axes. Les paramètres directement en relation avec ces options sont :

MODULX = DESS(46)
MODULY = DESS(48)

Ces options ne sont effectivement utilisables que dans le cas d'échelle linéaire sur l'axe correspondant.

Deux valeurs possibles,

INUMX = 0 (INUMY = 0) donné par défaut
INUMX = 1 (INUMY = 1)

INUMX = 0 (INUMY = 0)

La graduation inscrite sur le dessin sera la graduation réelle.

INUMX = 1 (INUMY = 1)

Dans ce cas, il faut alors préciser la valeur entière MODULX (MODULY) différente de 0.

Cette option n'est utile que lorsqu'on veut faire une graduation particulière de l'axe correspondant.

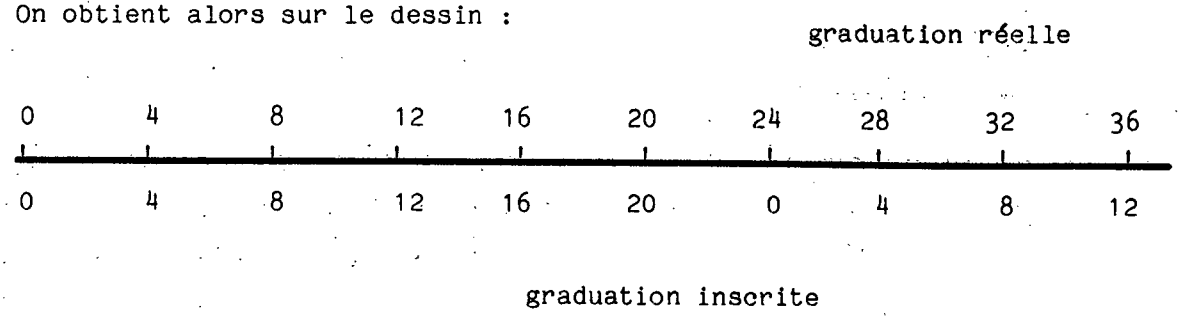
Donnons un exemple qui facilitera la compréhension.

L'axe des x représente une durée exprimée en heures de 0 à 72 h, et supposons que l'on veuille la graduation en heure jour par jour, avec inscription du temps toutes les 4 heures.

Cela est possible en choisissant par exemple :

IDON(3) = 18
IDON(5) = 4
INMUX = 1
MODULX = 24

On obtient alors sur le dessin :



5. PARAMETRES LIES A L'ECRITURE DE LEGENDES POUR LES COURBES

LEGCOU, XLONG, ILMULT, YLONG, NUCOU

A - LEGCOU = DESS(49), ILMULT = DESS(51)

Option permettant d'inscrire un commentaire sur le dessin par l'intermédiaire d'un sous-programme externe EC fourni par l'utilisateur. ILMULT permet de choisir l'emplacement de ce commentaire sur le dessin.

Les paramètres en relation avec ces options sont le sous-programme externe EC, les paramètres XLONG, YLONG, NUCOU (respectivement DESS(50), DESS(52), DESS(53)).

Deux valeurs pour LEGCOU

LEGCOU = 0 ← valeur donnée par défaut
LEGCOU = 1

et quatre valeurs pour ILMULT

ILMULT = 0 ← valeur donnée par défaut
ILMULT = 1
ILMULT = 2
ILMULT = 3

LEGCOU = 0

Pas d'écriture de légende. On ne fait évidemment pas appel au sous-programme EC (et l'utilisateur n'est pas obligé de le fournir). Il est aussi inutile de s'occuper des parametres ILMULT, XLONG, YLONG, NUCOU.

LEGCOU = 1

On fait effectivement appel au sous programme EC, qui devra se présenter sous la forme

```
      SUBROUTINE EC (HX, HY)
      ...
      ...
      ...
      RETURN
      END
```

HX et HY représentant la largeur et la hauteur des caractères. Exemple de sous-programme

```
      SUBROUTINE ECRI(HX,HY)
      COMMON/ECR/ITAB(5), NCARA, T, NFORMA
(*)   CALL PCARA (0., 0., 2, ITAB, NCARA, HX, HY, 1., 0.)
      CALL NOMBA (HX, 0., 2, T, NFORMA, HX, 1., 0.)
      RETURN
      END.
```

(*) le positionnement de la légende est calculé par le programme et il n'y a pas de déplacement relatif à faire lors de l'écriture du 1er cractère. Ce positionnement se fera de différentes façons selon la valeur de ILMULT.

IMULT = 0

On a la possibilité de référencer chaque courbe tracée. Après avoir tracé la courbe, on rappelle sur une longueur XLONG (donnée en centimètres sur le dessin initial -XLONG initialisé à 6. par défaut), le tireté constituant la courbe (ou bien le caractère spécial, lorsque la courbe est représentée par un symbole).

Le commentaire ne doit être écrit que sur une seule ligne (comme dans l'exemple donné précédemment). Le nombre de commentaires (ou légendes) que l'on peut mettre est calculé par le programme de façon à ne pas déborder du cadre objet. Dans le cas du cadre standard ISTAND = 1 ou 2, on peut mettre 8 courbes avec une légende pour chaque courbe. Si le nombre de courbes tracées par l'utilisateur est trop élevé, les dernières ne seront pas référencées. C'est évidemment l'option la plus commode pour l'utilisateur.

ILMULT = 1

Repère R_1 pour l'emplacement du commentaire.

Les paramètres utilisés pour déterminer le début d'écriture du commentaire sont XLONG, NUCOU.

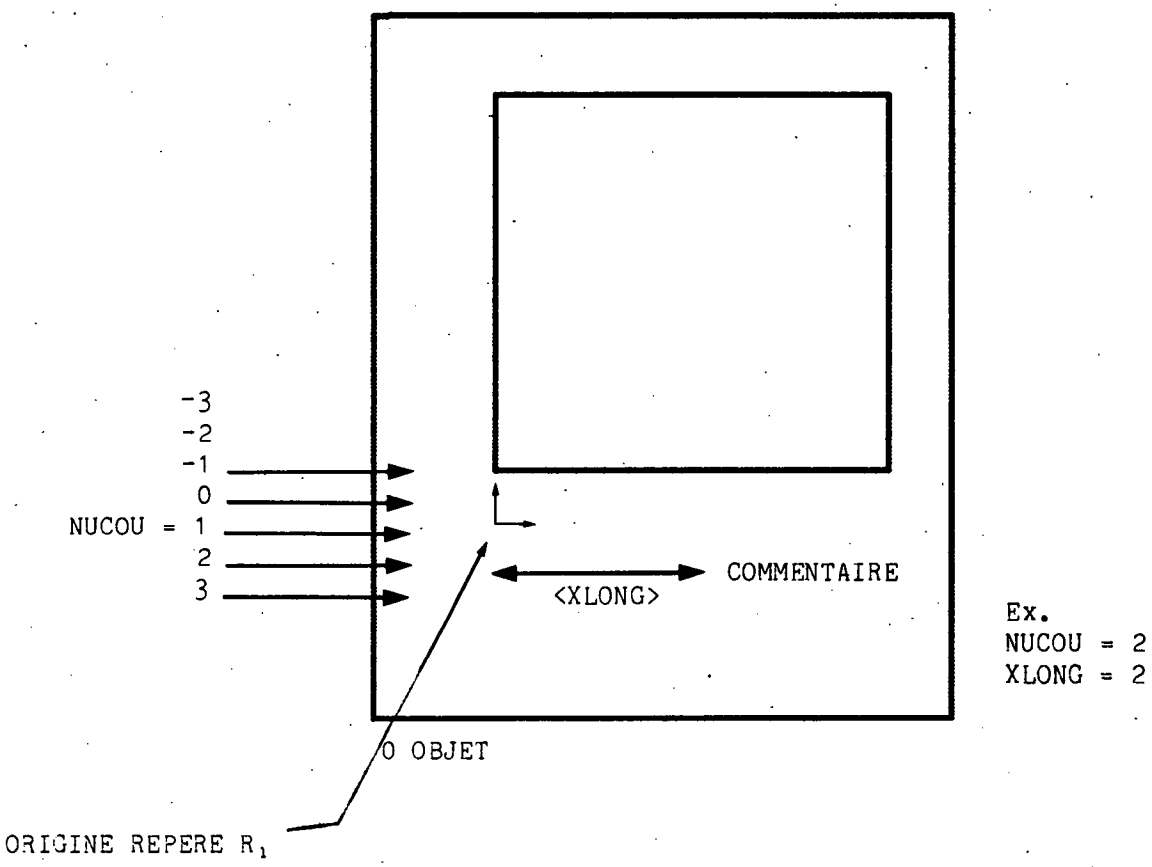
XLONG est donné en centimètres pour le dessin initial
NUCOU est un entier relatif (numéro de courbe).

L'origine du repère R_1 par rapport à l'origine OBJET du dessin est calculée par le programme de la façon suivante :

ABSCISSE : RDON(1)
ORDONNEE : RDON(3) - 6 x RDON(10) - RDON(11)

Dans le cas où IGRAD = 0 on ne retranche pas RDON(11). L'intervalle entre 2 NUCOU successifs est RDON(10) * 1.2375

Aucun test n'est fait sur l'emplacement du commentaire et l'on peut sortir du cadre OBJET si on ne prend pas garde.

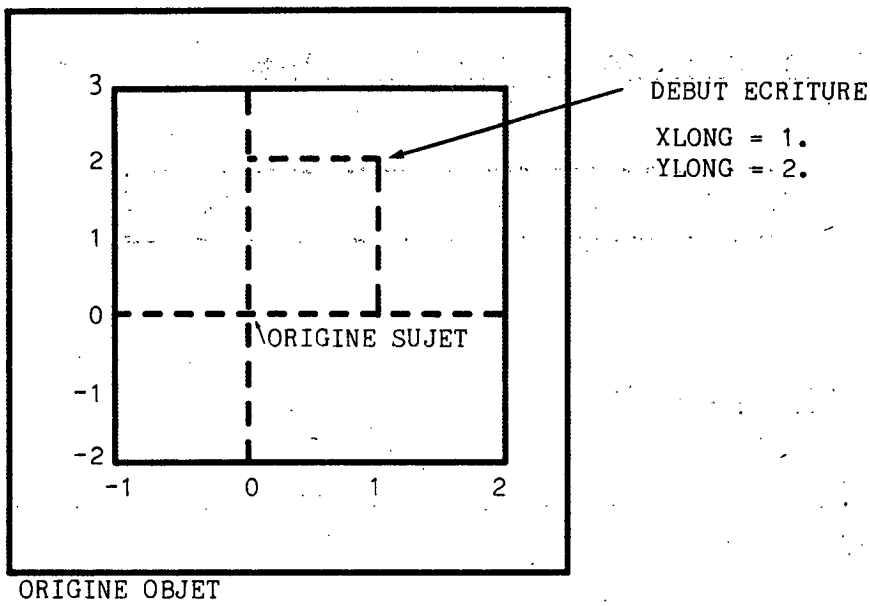


Ce repère R_1 est pris de façon implicite avec l'option ILMULT = 0.

ILMULT = 2

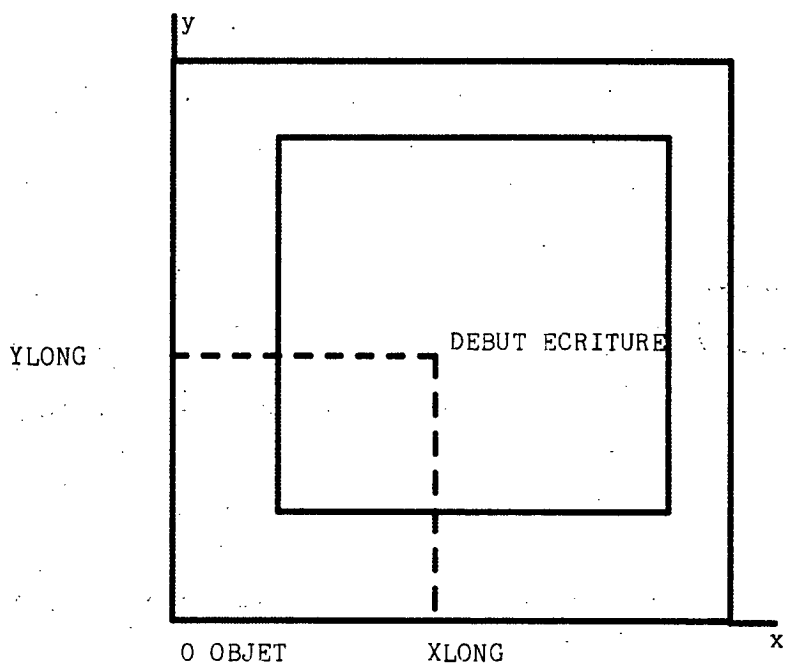
Le repère utilisé pour l'emplacement du commentaire est le repère SUJET et les coordonnées du début d'écriture sont données par XLONG, YLONG (unités sujet).

Comme dans le cas précédent, aucun test n'est fait sur l'emplacement de la légende, et l'on peut sortir du cadre sujet et même du cadre objet si les valeurs de XLONG et YLONG sont mal choisies.



ILMULT = 3

Le repère utilisé cette fois-ci pour l'emplacement du commentaire est le repère OBJET et les coordonnées du début écriture du commentaire sont toujours données par XLONG eet YLONG exprimée en centimètres (sur le dessin initial).



6. PARAMETRES RELATIFS AU TRACE DE LA COURBE

ISUPE, ICHOIX, ICHAR, DIME, PAS(10), NP

A - ISUPE = DESS(54)

Option offrant la possiblité de tracer plusiseurs courbes dans le même dessin (même cadre sujet).

Deux valeurs pour cette option

ISUPE = 0 ← valeur donnée par défaut
ISUPE = 1

ISUPE = 0

A chaque appel au traceur avec ISUPE = 0 (en fait une valeur ≠ 1) un nouveau dessin sera fait, avec tout ce que cela comporte : tracé des cadres, des graduations, des légendes, ... etc.

ISUPE = 1

Seule la courbe est tracée (avec légende éventuellement) dans le cadre sujet du dessin PRECEDENT, avec les mêmes échelles sur les axes. Le premier appel au traceur doit être fait avec ISUPE = 0.

Pour les différentes courbes successives on n'aura sur le dessin que la partie visible dans le cadre sujet.

B - ICHAR = DESS(56)

Options relatives au tracé de la courbe.

Les paramètres en relation sont

DIME = DESS(57)
PAS = DESS(58:67)
NP = DESS(68)
ICHOIX = DESS(55)

Deux valeurs pour ICHAR

ICHAR = 0 (≠ 1) donnée par défaut
ICHAR = 1

ICHAR = 0

La courbe est tracée de tirets (trait continu, tireté, pointillé, ...)

Deux possibilités de choix : choisir un tireté élaboré par le programme ICHOIX > 0, ou décrire soi-même le tireté lorsque ICHOIX ≤ 0 à l'aide du tableau PAS et de l'entier NP.

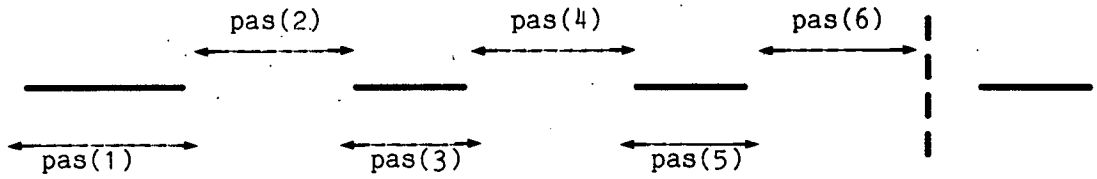
- Cas ICHOIX > 0

ICHOIX = 1	trait continu	————
ICHOIX = 2	pointillé 1	.....
ICHOIX = 3	pointillé 2	- - - - -
ICHOIX = 4	tireté 1	-·-·-·-
ICHOIX = 5	tireté 2	—·—·—·—
ICHOIX = 6	tireté 3	——·——·——
ICHOIX = 7	tireté 4	———·———

Si ICHOIX > 7 la valeur est prise modulo 7.

- Cas ICHOIX ≤ 0

L'utilisateur définira lui-même le tireté à l'aide de NP (pair) valeurs du tableau PAS, sachant que les "tirets" de numéro impair sont effectivement tracés sur le dessin, ceux de rang pair étant "tracés plume levée".



Les valeurs des éléments du tableau PAS sont données en centimètres, et sont valables pour le dessin initial (c'est-à-dire qu'elles sont affectées par les paramètres d'homothétie).

ICHAR = 1

La courbe est repérée par des symboles centrés.

En chaque point de la courbe VX(I) VY(I), on trace un symbole caractérisé par ICHOIX et dont la grandeur est donnée par DIME en centimètres, sur le dessin initial.

ICHOIX = 1	+	(défaut)
ICHOIX = 2	×	
ICHOIX = 3	*	
ICHOIX = 4	□	
ICHOIX = 5		
ICHOIX = 6	Δ	
ICHOIX = 7	∇	
ICHOIX = 8	◇	
ICHOIX = 9	o	

7. PARAMETRES RELATIFS AU DESSIN FINAL

HOMX, HOMY, NBLOC, NUNIT, IID

Ce common apparaitra dans tous les modules de la bibliothèque.

A - HOMX = DESS(69), HOMY = DESS(69)

Ces paramètres agissent sur les dimensions du dessin initial. Le dessin initial subit une transformation avant affichage

- une homothetie de rapport HOMX dans la direction x
- une homothetie de rapport HOMY dans la direction y

Le centre d'homethétie est l'origine OBJET.

Pour que le dessin initial soit aussi le dessin final, il suffit de prendre HOMX = HOMY = 1.

Attention au choix des valeurs HOMX, HOMY, le dessin final celui effcitivement tracé ne doit être ni trop petit, ni trop grand.

Ainsi, par exemple avec les initialisations retenues dans le cas ISTAND =1 ou 2, le dessin initial a une dimension 33,6 cm × 47,5 cm et ainsi on ne pourra pas prendre HOMY > 1.47 sinon le tracé sort des limites du papier BENSON (largeur 70 cm).

Dans le cas ISTAND = 0, les valeurs définissant le dessin et fournies par l'utilisateur correspondent bien entendu à HOMX = HOMY = 1.

Les valeurs par défaut sont :

HOMX = .5
HOMY = .5

si bien que le dessin final a pour dimensions 16,8 × 23,75 cm.

B - NBLOC = DESS(71)

Numéro de bloc BENSON -initialisé à 1 par défaut.

L'utilisateur n'a pas à agir sur cette variable.

Les numéros de blocs correspondant à telle ou telle partie du dessin sont reportés sur le listing si ILIST = 1.

C - NUNIT = DESS(72)

Numéro logique de l'unité de sortie.

Initialisé à 10 par défaut.

APPENDICE B

OPTIMISATION

par JF BONNANS

TABLE DES MATIERES

I - INTRODUCTION

Les algorithmes d'optimisation implantés actuellement permettent de résoudre des problèmes de type sans contraintes

(1) $\min f(x)$

ou avec des contraintes de borne

(2) $\min f(x)\ ;\ binf(\ ) \leq x(\ ) \leq bsup(\ ),$

les inégalités ci-dessus étant prises pour chaque composantes de la matrice x. Le critère f est supposé régulier.

L'utilisateur doit fournir un module appelé simulateur, écrit en FORTRAN ou en Blaise, qui fournit pour chaque x :

- la valuer de f(x) et/ou de son gradient,
- éventuellement, un signal de sortie du domaine de définition de f.

Les codes d'optimisation sont extraits de la bibliothèque Modulopt, développée à l'INRIA. Les normes d'écriture de simulateurs écrits en FORTRAN sont compatibles avec celles de Modulopt.

Les principaux algorithmes d'optimisation sont exposés de manière synthétique dans Bertsekas (1982), Gill-Murray-Wright (1981), Minoux (1983).

Développements envisagés dans un avenir proche

- Implémentation d'algorithmes pour les fonctions non différentiables, avec ou sans contraintes de borne.

- Implémentation d'un algorithme de lagrangien augmenté.

Les critiques et suggestions seront prises en compte autant que possible dans le développement des nouvelles versions.

II - MINIMISATION SANS CONTRAINTES

II.1 - Principes généraux
II.2 - Méthodes de quasi-Newton
II.3 - Problèmes de grandes tailles

II.1 - Principes généraux

Dans toute la suite, on considèrera la matrice x de dimension $n_1 \times n_2$ comme un vecteur de dimension $n = n_1 \times n_2$, afin de simplifier les notations. Cette section expose les principes généraux des méthodes de descente lorsque le critère est régulier.

On appelle gradient de f en x le vecteur

$$\nabla f(x) = (\partial f / \partial x_1, \ldots, \partial f / \partial x_n)^t.$$

Si $\nabla f(x) \neq 0$, on dit que d est une direction de descente en x si $d^t \nabla f(x) < 0$. Ceci assure que pour un certain $t_0 > 0$, $f(x + td) < f(x)$, $\forall 0 < t < t_0$.

Un moyen de calculer une direction de descente est de choisir M symétrique, définie positive et de poser $d = -M\nabla f(x)$. Le schéma général des algorithmes est

0) Choisir $x^0 \in \mathbf{R}^n$; $k = 0$.
1) Si $||\nabla f(x^k)|| < \varepsilon$, arrêt ; sinon, aller en 2).
2) Choisir une direction de descente d^k.
3) Choisir $t^k > 0$ tel que $f(x^k + t^k d^k) < f(x^k)$.
4) $k = k + 1$; aller en 2).

L'étape 3 s'appelle la recherche linéaire. La règle de recherche linéaire la plus répandue est la :

Règle de Wolfe

Choisir m_1, m_2 tels que $0 < m_1 < 1$, $1/2 \leq m_2 < 1$ et trouver t^k tel que

W1 $\quad f(x^k + t^k d^k) \leq f(x^k) + t^k\, m_1\, \nabla f(x^k)^t d^k$,

W2 $\quad \nabla f(x^k + t^k d^k)^t\, d^k \geq m_2\, \nabla f(x^k)^t\, d^k$. □

Notons que W1 assure la décroissance du critère alors que W2 empêche le pas t^k d'être arbitrairement petit.

Sous des hypothèses convenables, les méthodes de descente convergent vers un point stationnaire (c'est-à-dire de gradient nul).

Si ce point est un minimum local, son hessien est positif. S'il est défini positif, notons C son conditionnement. La méthode du gradient (c.a.d. avec $d^k = -\nabla f(x^k)$) associée à la règle de Cauchy (t^k optimal) converge linéairement, à la vitesse $(C-1)/(C+1)$. Pour toutes les méthodes, un mauvais conditionnement du hessien provoquera des difficultés numériques (convergence lente, overflow). Notons qu'un problème mal conditionné est un problème pour lequel une petite perturbation des données peut changer considérablement la solution ; il est donc mal posé. En conséquence, il faut essayer lors de la form lation du problème d'optimisation d'obtenir un problème bien conditionné. Pour ceci on conseille de mettre les variables à l'échelle, c.a.d. de faire en sorte qu'elles aient toutes le même ordre de grandeur.

Les tests d'arrêt de l'algorithme peuvent porter sur :

- la norme du gradient,
- la décroissance du critère,
- les variations de la variable x.

II.2 - Méthodes de quasi Newton (optimiseur 10)

Dans cette méthode, on stocke une approximation du hessien B^k. La remise à jour de B^k se fait en utilisant l'information en x^k et x^{k-1} ; si on note

$$s^k = x^k - x^{k-1} \ , \ y^k = \nabla f(x^k) - \nabla f(x^{k-1}),$$

la formule de BFGS est

$$B^k = B^{k-1} + y^K(y^k)^t \ / \ (S^K)^t y^k - B^{k-1}S^k(B^{k-1}S^k)^t \ / \ (S^k)^t \ B^{k-1} \ S^k.$$

En pratique on stocke le facteur de Choleski de B^k dont la remise à jour s'effectue à l'aide de celle de B^{k-1}, la règle de recherche linéaire étant de type Wolfe. Le code est issu de celui de M.J.D. Powell (1976) où on trouvera une description détaillée de l'algorihme.

Avantages de la méthode : Cet algorithme est le plus rapide de ceux utilisant seulement l'information sur les gradients. Dans de nombreux cas, ses performances sont de l'ordre de la méthode de Newton.

Inconvénients : Il y en a deux :

- Le stockage du facteur de Choleski de B^k nécessite $O(n^2)$ mémoires. La méthode est donc impraticable si n est trop grand.

- L'approximation du hessien à partir du gradient peut engendrer des instabilités se traduisant par un mauvais conditionnement de B^k. L'algorithme peut alors se bloquer en particulier si des erreurs numériques s'introduisent dans le calcul du gradient.

II.3 - Problèmes de grande taille

Les algorithmes les mieux adaptés aux problèmes de grande taille sont de type gradient conjugué ou, mieux, quasi-Newton à mémoire limitée (qNml). En ajoutant des contraintes de borne au problème, on peut utiliser l'algorithme qNml pour les contraintes de borne implanté dans Blaise (voir §III).

III - MINIMISATION AVEC CONTRAINTES DE BORNES

On expose les algorithmes de résolution de problèmes du type

$$\min f(x) \; ; \; a(i) \leq x(i) \leq b(i), \; i = 1 \text{ à } n.$$

III.1 - Méthode de quasi Newton avec projection

On notera P la projection sur l'ensemble vérifiant les contraintes de bornes. On sait que

$$P(x)(i) = \max (a(i), \min(x(i), b(i))) \; ; \; i = 1 \text{ à } n.$$

Soit B^k l'approximation du hessien obtenu pas la méthode de BFGS (voir §II). A l'étape k de l'algorithme, on considère une prédiction de l'ensemble des indices actifs I_B^k, contenant les varirables localement bloquées à leur borne compte-tenu du signe du gradient. Dans l'ensemble des variables libres $I_L^k = (1,n) - I_B^k$) on calcule $B^{\overline{k}}$, la restriction de B^k, et $d^{\overline{k}} = -(B^{\overline{k}})^{-1} \nabla f(x^k)$. Puis on prend

$$x^{k+1} = P(x^k + \rho^k d^k),$$

où ρ^k est donné par une recherche linéaire convenable. L'algorithme peut se formaliser comme suit :

0) Choisir x^0 admissible, $k = 0$, $B^0 = Id$.

1) Calculer une matrice symétrique définie positive à partir de B^k et $d^k = -(M^k)^{-1} \nabla f(x^k)$.

2) Choisir $\rho^k > 0$ et $x^{k+1} = P(x^k + \rho^k d^k)$.

3) $k = k+1$. Poser $s^k = x^k - x^{k-1}$, $y^k = \nabla f(x^k) - \nabla f(x^{k-1})$ et

$$B^k = B^{k-1} + y^k (y^k)^t / (s^k)^t y^k - B^{k-1} S^k (B^{k-1} s^k)^k / (s^k)^t B^{k-1} S^k$$

4) Test d'arrêt. Eventuellement, aller en 1).

Remarque

En réalité l'algorithme est un peu plus compliqué :

- Une modification de la formule de BFGS (inactive si les contraintes ne sont pas actives) préserve la stricte positivité de B^k.

- Pour éviter le zig-zag (saturation et libération alternées d'une contrainte) qui freine la convergence, l'algorithme effectue une succession de cycles au cours desquels aucune variable bloquée n'est relachée.

Avantages et inconvénients : Pour la méthode de BFGS, voir le §II. En particulier la méthode est inutilisable pour n trop grand. La méthode de projection permet de saturer comme de libérer plusieurs contraintes à la fois. Le découpage en cycle est un procédé efficace pour éviter le zig-zag.

III.2 - Méthode pour les systèmes de grande taille

La méthode utilisée est du type précédent, la matrice B^k étant calculée par une méthode de quasi-Newton à mémoire limitée, ce qui permet de ne stocker que quelques vecteurs de dimension n (voir Shanno (1978)).

IV - MINIMISATION SOUS DES CONTRAINTES GENERALES

On considère un problème du type

$$\min f(x),\ g_i(x) = 0,\ i = 1 \text{ à } ne,$$
$$g_i(x) \leq 0,\ i = ne + 1 \text{ à } nc$$

avec $0 \leq ne \leq nc$ et f,g régulières.

Aucun algorithme n'est implanté pour l'instant, qui puisse résoudre cette classe de problème (une méthode de lagrangien augmenté sera implémentée prochainement). Néanmoins, les outils actuels donnent le moyen d'aborder certains de ces problèmes. A titre d'exemple, le simulateur étant écrit aux normes décrites au §5, voici un exemple de simulateur calculant le critère pénalisé et son gradient.

```
//<fpen,gpen,ind>=sipn(x,ind);
// calcul d'un critere penalise et de son gradient
// attention les variables ne,nc,cpen doivent etre initialisees
//           le simulateur appele sip1 doit etre aux normes de la doc
<f,g,indic>=sip1(x,ind);
if indic < 0 then ind=indic, return, end;
if nc > ne then f(ne+1:nc)=ppos(f(ne+1:nc)),end;
fpen=f(nc+1) + cpen*norm(f(1:nc))**2/2;
if ind=2 then return,end;
gpen=g(:,nc+1);
if ne > 0 then
   for i=1:ne, gpen=gpen + cpen*f(i)*g(:,i),end,end;
if nc > ne then
   for i=ne+1:nc, if f(i) > 0 then gpen=gpen + cpen*f(i)*g(:,i),end,end,end;
```

(Attention : on utilise ici la fonction ppos qui donne la partie positive d'une matrice, élément par élément).

Si sip1 est le simulateur écrit au §5 on peut résoudre le problème pénalisé correspondant en appelant

```
//<f,x,g>=tpen(x0,nap,cpen1);
//test du probleme penalise
cpen=cpen1, ne=1,nc=2;
<f,x,g>=optim(10,nap,x0,sipn,%eps);
```

On peut de la même manière écrire un simulateur permettant de calculer un lagrangien augmenté et son gradient, et écrire l'algorithme du langrangien augmenté en appelant à chaque itération un des optimiseurs déjà implantés.

V - NORMES D'ECRITURE DU SIMULATEUR

V.1 - Simulateur Blaise

La liste d'appel sera de la forme

```
//<F,G,IND>=SIMUL(X,IND)
```

où X est la variable à optimiser
F,G, le critère et son gradient.

La variable IND a pour signification en entrée :

IND =	1	impression au gré de l'utilisateur
	2	calcul de F
	3	G
	4	F et G

et en sortie :

IND <	0	F(X) non défini
=	0	arrêt des calculs demandé par le simulateur
>	0	calcul correctement effectué.

Exemple 5.1

Soit $F(X) = 1/2\ X^TAX - BX$ où A est symétrique. Le simulateur sera :

```
//<F,G,IND>=SIMUL<X,IND>
// remarque : dans le calcul-dessous on peut ne calculer a*x qu'une fois
  if (IND-2)*(IND-4)=0 then F=X'*A*X/2-B*X,end,
  if (IND-3)*(IND-4)=0 then G=A*X-B,end;
```

Exemple 5.2

Fonction de Rosenbrock (voir par exemple Gill-Murray-Wright (1981).

```
//<F,G>=ROSEN(X)
      A=X(2)-X(1)**2
      B=1-X(2)
      F=100**A**2+B*2
      G(1)=-400*A*X(1)
      G(2)=200*A-2*B
```

V.2 - Simulateur FORTRAN

La liste d'appel sera de la forme

```
SIMUL(IND,N,X,F,G,IZS,RZS,DZS)
```

avec

X,F,G, comme ci-dessus
N dimension de X
IZS,RZS,DZS : tableaux de travail respectivement entiers, simple et double précision
IND comme ci-dessus, avec de plus :

IND = 10 le simulateur initialise NIZS,NRZS,NDZS, les dimensions de IZS,RZS,DZS qui se trouvent dans le common

```
COMMON/NIRD/NIZS,NRZS,NDZS
```

IND = 11 le simulateur initialise IZS,RZS, DZS.

L'initialisation des paramètres, commandée par la présence de "in" dans la liste d'appel de optim, se fera donc en appelant successivement le simulateur avec INDIC = 10 (la place mémoire est alors réservée dans la pile de Blaise) puis INDIC = 11.

Exemple 5.3

Reprenons la fonction de Rosenbrock et mettons le coefficient 100 en paramètre.

```
SIMUL(IND,N,X,F,G,IZS,RZS,DZS)
IMPLICIT DOUBLE PRECISION (A-H,O-Z)
DIMENSION X(1),G(1),IZS(1),RZS(1),DZS(1)
COMMON/NIRD/NIZS,NRZS,NDZS
IF (IND.EQ.10) THEN
   NIZS = 1
   NRZS = 1
   NDZS = 3
   RETURN
ENDIF
```

```
IF (IND.EQ.11) THEN
   DZS(1) = 100.D0
   DZS(2) = 2.D0 * DZS(1)
   DZS(3) = 4.D0 * DZS(1)
   RETURN
ENDIF
A = X(2)-X(1)**2
B = 1 - X(2)
F = DZS(1)*A**2+B**2
IF (IND.EQ.2) RETURN
G(1) = -DZS(3)*A*X(1)
G(2) = DZS(2)*A-2.D0*B
RETURN
END
```

V.3 - Simulateurs pour les problèmes avec contraintes

On considère ici le problème général du §4. Il n'existe pas pour ce problème de standard indiscutable. En effet, certaines méthodes ne nécessitent que le gradient des contraintes actives (ou supérieures à $-\lambda_j/2c$ pour les méthodes de lagrangien augmenté), alors que, dans les problèmes de contrôle avec des contraintes sur l'état traitées par dualité, il est absurde de fournir les gradients des contraites un par un. De plus, il faudrait séparer le traitement des contraintes linéaires. Ceci est contraignant pour celui qui a la charge de l'écriture du simulateur. On se contentera donc du standard suivant : le simulateur fournit tous les gradients, quitte à programmer soi-même un algorithme si on veut éviter de calculer les gradients des contraintes séparément (le cas du lagrangien augmenté est traité en IV). Plus précisément, un simulateur écrit en Blaise sera de la forme

<F,G,IND> = SIMUL(X,IND),

où IND est comme dans le cas sans contraintes, et

F(I),G(:,I) valeur et gradient de la contrainte numéro I (I=1 à NC)

F(NC+1),G(:,NC+1) valeur et gradient du critère.

Un simulateur Fortran sera de la forme

```
SIMUL(IND,N,X,F,G,IZS,RZS,DZS)
```

où F et G sont comme ci-dessus et (IND,N,IZS,RZS,DZS) sont comme dans le cas sans contraintes.

Exemple 5.4

Soit le problème

$$\min (x_1^2 + x_2^2) / 2 \; ; \; x_1 + x_2 - 1 \leq 0, \; x_1 \geq 0.$$

Le simulateur Blaise correspondant sera (nous l'appellerons SIP1)

```
//<F,G,IND>=SIP1(X,IND);
F= < X(1)+X(2)-1, -X(1), (X(1)**2+X(2)**2)/2>;
G= < 1, -1, X(1);
     1,  0, X(2)>;
```

Remarque

Dans le futur, des algorithmes seront probablement implantés qui :

- effectuent un traitement séparé pour les contraintes de borne, linéaires et non linéaires. Dans ce cas on fournit une fois pour toutes à l'algorithme la matrice et le second membre des contraintes linéaires,

- et/ou traitent séparément les variables interviennent linéairement dans les contraintes et le critère.

VI - **PERSPECTIVES**

Seront implantés à court terme :

- Des algorithmes de descente pour la minimisation de fonctions non différentiables, avec ou sans contraintes de borne. On pourra les appliquer en particulier aux problèmes de minimax. Ils permettront de traiter des contraintes générales par pénalisation exacte.

- Un algorithme de lagrangien augmenté.

VII - **BIBLIOGRAPHIE**

D.P. BERTSEKAS (1982). Constrained optimization and Lagrange multipliers methods. Academic Press, New York.

J.F. BONNANS (1983). A variant of a projected variable metric method for bound constrained optimization problem. Rapp. INRIA n° 242.

J.E. DENNIS, J.J. MORE (1977). Quasi-Newton methods, motivation and theory. SIAM Review 19, p. 46-89.

P.E. GILL, W. MURRAY, M. WRIGHT (1981). Practical optimization. Academic Press, New York.

M. MINOUX (1983). Programmation mathématique, théorie et algorithmes. Dunod, Paris 1983.

M.J.D. POWELL (1976). Some global convergence properties of a variable metric algorithm for minimization without exact line searches. in Non-linear Programming, SIAM-AMS Proc., Vol. 9, An? Math. Sci. Providence, R.I.

D.F. SHANNO (1978). Conjugate gradient methods with inexact line searches. Math. Op. Res., p. 224-256.

P. WOLFE (1969). Convergence conditions for ascent methods. SIAM Review 11, p. 226-235.

APPENDICE C

SIMULATION

TABLE DES MATIERES

SIMULATION
(Commandes Ode et Impl)

Les algorithmes implantés permettent de résoudre un système de N équations différentielles donné sous forme explicite :

$$y' = f(t,y)$$

ou implicite

$$a(t,y)y' = g(t,y).$$

La méthode de résolution change selon que le système est raide (présence de modes rapides) ou non raide.

L'utilisateur doit spécifier la méthode d'intégration choisie (méthode dite "BDF" (Backward Differentiation Formulas) pour les systèmes raides, méthode d'Adams pour les systèmes non raides. Cependant si l'utilisateur ignore le type de son problème la sélection peut se faire automatiquement.

METHODES D'INTEGRATION

Soit une grille de discrétisation $(t_0, t_1, \ldots, t_n, \ldots)$ et $h = t_n - t_{n-1}$ fixé et y_n l'approximation de $y(t_n)$ cherchée pour y_0 donné.

Pour les systèmes non raides on utilise les formules implicites d'Adams :

$$(1) \qquad y_n = y_{n-1} + h \sum_{i=0}^{q-1} \beta_i \, y'_{n-i} \,, \quad y'_k = f(t_k, y_k)$$

Le système est implicite puisque $\beta_0 > 0$.
q est l'ordre du schéma ($1 \leq q \leq 12$) de discrétisation et les coefficients β_i ne dépendent que de q.

La solution de ce problème est obtenue par approximations successives :

$$(2) \qquad y_n(m+1) = y_{n-1} + h\,\beta_0\, f(t_n, y_n(m)) + h \sum_{i=1}^{q-1} \beta_i\, y'_{n-i}.$$

Le pas h et l'ordre q varient durant le processus d'intégration en fonction des estimées des erreurs locales commises et des tolérances fournies par l'utilisateur. Notons qu'aucune matrice N × N n'apparait ici.

Pour les systèmes raides les formules d'approximation sont :

$$(3) \qquad y_n = \sum_{i=1}^{q} \alpha_i\, y_{n-i} + h\,\beta_0\, y'_n$$

$$= a_n + h\,\beta_0\, f(t_n, y_n).$$

Ici l'ordre q vérifie $1 \leq q \leq 5$.

Le schéma d'itérations précédent peut ne pas converger en raison de la forte dépendance de f par rapport à y. On utilise donc une méthode de type Newton

$$(4) \qquad P(y_n(m+1) - y_n(m)) = y_n(m) - a_n - h\,\beta_0\, f(t_n, y_n(m))$$

où m est toujours l'indice d'itération.
P est une matrice N × N approximant le Jacobien du système non linéaire qu'on résout

$$P \simeq I - h\,\beta_0\, J,\ J = \text{matrice jacobienne de f par rapport à y.}$$

La différence avec une méthode de Newton "exacte" est que J est évaluée seulement périodiquement et la même valeur de P est utilisée pour toutes les itérations d'un pas de temps donné. Ici aussi h et q peuvent varier en fonctin des tolérances fournies par l'utilisateur.

La matrice J est calculée par un programme fourni par l'utilisateur ou alors évaluée par différences finies.

Les systèmes implicites sont intégrés en multipliant les deux membres de (1) ou (3) par $A(t_n, y_n)$, en remplaçant $A(t_n, y_n)y'_n$ par $g(t_n, y_n)$ et en résolvant le système implicite obtenu par un schéma de type (2) ou (4), ce dernier schéma étant en général préférable.

Notons que la matrice A peut être singulière mais l'utilisateur doit fournir des valeurs consistentes de y_0 et y'_0. Dans ce cas le système résolu est algébrico-différentiel et l'utilisateur doit être prudent pour formuler un problème bien posé.

DEMARRAGE A CHAUD

Il peut être utile d'appeler les commandes d'intégration pour certaines valeurs de t, de regarder les résultats puis de vouloir calculer la solution pour une (ou plusieurs) valeurs de t, soit $t_1 > t$ sans avoir à refaire tous les calculs.

Cela est possible pour la commande ode par l'intermédiaire de deux tableaux de travail optionnels w et iw.

La commande Blaise sera donc

```
<y,w,iw> = ode(...);
```

Pour sauver w et iw, et pour un redémarrage à chaud on utilise alors la commande

```
ode(..., t, ..., w, iw)
```

en prenant pour valeur "initiale" y_0 la dernière valeur calculée et pour instant initial la valeur t correspondante.

Les vecteurs w et iw contiennent de l'information qui peut être utile. En particulier notons :

w(5) = premier pas d'intégration effectivement utilisé,
w(11) = dernier pas d'intégration effectivement utilisé,
w(12) = pas d'intégration qui sera tenté au prochain calcul,
w(13) = dernière valeur de t où le calcul de y a été effectué par le solveur
w(15) = dernière valeur de t pour laquelle le solveur est passé d'une méthode d'intégration à l'autre. (Valable seulement dans le cas où la méthode d'intégration n'est pas précisée).

iw(11) = nombre de calculs de y effectués par le solveur,
iw(12) = nombre d'évaluation de f,
iw(13) = nombre d'évaluation du jacobien,
iw(14) = ordre courant de l'approximation,
iw(15) = ordre tenté au prochain calcul,
iw(19) et iw(20) indiquent respectivement la dernière méthode d'intégration utilisée (1 = Adams, 2 = raide) et celle tentée pour le prochain calcul. (Valable seulement si la méthode d'intégration n'est pas précisée).

Pour sauver l'état du solveur d'une session Blaise sur l'autre, il faut, en plus de w et iw, sauver et restaurer le contenu de common FORTRAN. Cela peut se faire à l'aide des deux macros suivantes.

Cas des solveurs où la méthode d'intégration est précisée

```
||<x,y> = svcom
||sauvegarde
  <x,y> = fort('svcom', 'sort', <1,219>, 1, 'd', <1,41>, 2, 'i')

||rscom(x,y)
||restauration
  fort('rscom', x, 1, 'd', y, 2, 'i', 'sort')
```

Ces deux macros sont équivalentes aux programmes Fortran suivants :

```
      dimension x(219), ix(41)
      double precision x
      call svcom (x,ix)
c     write x, ix sur fichier pour sauvegarde
      stop
      end

      dimension x(219), ix(41)
      double precision x
c     read x, ix sur fichier pour restauration
      .
      .
c     passage de x et ix au solveur
      call svcom(x,ix)
      stop
      end
```

Cas du solveur où la méthode d'intégration est automatique

Les macros Blaise correspondantes sont :

```
||<x,y> = svcma
<x,y) = fort ('svcma', 'sort', <1,241>, 1, 'd', <1,50>, 2, 'i')

||rscma(x,y)
fort ('rscma', x, 1, 'd', y, 2, 'i').
```

Ici les commons double précision et entrée sont donc de tailles 241 et 50 respectivement.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique